AFRL-IF-RS-TR-2002-139
Final Technical Report
June 2002

DYNAMIC SECURITY ANALYSIS OF COSTS APPLICATIONS

Reliable Software Technologies

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. F021

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

AFRL-IF-RS-TR-2002-139 has been reviewed and is approved for publication.

John C. Faust

APPROVED: JOHN C. FAUST
Project Engineer

[illegible]

FOR THE DIRECTOR: WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** Jun 02	**3. REPORT TYPE AND DATES COVERED** Final Aug 97 - Aug 99

4. TITLE AND SUBTITLE DYNAMIC SECURITY ANALYSIS OF COTS APPLICATIONS	**5. FUNDING NUMBERS** C - F30602-97-C-0117 PE - 62301E/33140F PR - 2301 TA - 01 WU - 05
6. AUTHOR(S) Matthew Schmid, Anup Ghosh and Frank Hill	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Reliable Software Technologies 21351 Ridgetop Circle Suite 400 Dulles, VA 20166	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 N. Fairfax Dr Arlington, VA 22203-1714 AFRL/IFGB 525 Brooks Rd Rome, NY 13441-4505	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2002-139

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: John Faust, IFGB, 315-330-4544

12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 words)*

This is the final technical report for a research effort whose objective was to analyze the robustness of Commercial Off-the-Shelf (COTS) software on the Windows NT platform. Robustness to anomalous data plays an integral role in the security of mission-critical COTS software. This report describes the experiments conducted, analysis of results, and prototypes built during the duration of the project. The bulk of this report discusses the three subprojects that formed the core of this research: The Random and Intelligent Data Design Library Environment (RIDDLE), NetHose, and the Failure Simulation Tool (FST). RIDDLE is a testing framework that provides the ability to perform automated testing of command-line utilities, Application Programming Interfaces (APIs), and network daemons. NetHose is a utility that can be used to test the robustness of the network stack. Both RIDDLE and NetHose rely on the use of a special input generation technique developed for this project. The FST is a prototype tool that allows a tester to analyze the robustness of a COTS application to the failure of operating system calls. The rationale, design, and use of each tool are discussed.

14. SUBJECT TERMS Information Warfare, Automated Testing, Software Robustness, Denial of Service Attack, Test Data Generation	**15. NUMBER OF PAGES** 62
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

LIST OF FIGURES

Executive Summary

In this project, we examined a key property of secure and dependable systems: robustness. Our research focused on developing techniques for analyzing as well as increasing the robustness of software due to unknown and anomalous events. The project was partitioned into three distinct and complementary threads of robustness research: intelligent test case generation, testing of the Win32 network stack for resilience to denial of service, and testing application software for robustness to failing operating system (OS) resources.

The Random and Intelligent Data Design Library Environment (RIDDLE) was developed to support intelligent black-box testing of operating system and command-line utilities to unusual inputs. The work leveraged research in robustness testing of OS utilities previously performed on Unix systems, most notably by Barton Miller's research group at the University of Wisconsin. Robustness testing of OS utilities has previously found vulnerabilities in operating system software to unexpected, random input. Adopting this approach for the Win32 platform, we enhanced it further by intelligently combining valid input with anomalous input in order to unmask flaws that remain hidden to purely random testing. RIDDLE provides a test harness and library for automatically generating test data according to both random (generally invalid) and valid parameters. Our studies have shown empirically the benefit derived from combining intelligent test data generation with random test case generation for the purpose of testing robustness.

Drawing on the experience of RIDDLE, we developed NetHose; a tool for testing the robustness of the Win32 network stack to anomalous data. The importance of this work is in developing structured approaches to testing critical portions of software that comprises the National Information Infrastructure (NII). Since, the network stack of the Win32 platform is a key component of the NII, this approach is among the first to independently and systematically study the robustness of this software under anomalous input conditions. Fundamental problems in the design of network protocols such as TCP/IP can leave all platforms vulnerable. However, flaws, in the networking software can leave a given platform vulnerable to anomalous use of network services, or attack. NetHose is a network stack testing utility that is able to test the network stack of Win32 systems to unexpected data. This type of analysis effectively tests the operating system's ability to handle unusual network packets. The approach uses combinations of valid and invalid data in packet header fields in order to test the robustness of the network stack against unusual packet headers. Our studies revealed three types of robustness failures: kernel exceptions (colloquially known as blue screen of death), hard freezes, and system slowdowns.

The third thread of research in this project was concerned with testing the robustness of Win32 applications under failing OS conditions. From our previous research with RIDDLE, we found that the three core libraries that compose the Win32 system more often than not throw memory access violation exceptions when presented unusual input. Thus, if application developers (particularly for mission-critical applications) do not account for these exceptions that the OS throws, then the application is likely to crash. The Failure Simulation Tool (FST) provides the ability to test Win32 executables (without requiring source code) for robustness to exceptions or errors returned by OS functions. FST instruments the interface between an application and the OS DLL in order to return errors or exceptions from an OS function. This is a far more efficient approach than black-box testing of an application in hopes of generating an OS exception. Experiments showed that Microsoft desktop applications had varying levels of non-robustness to exceptions and errors returned by OS functions. This type of non-robust behavior is typically expected from desktop applications. However, non-robustness to errors or exceptions returned from OS functions is typically not acceptable in a mission-critical application, such as a ship propulsion system. Thus, FST provides the ability to test mission-critical software for robustness to failing OS functions. In the final stage of this work, we used the instrumentation layer to provide protective wrappers for applications such that an exception can be caught by the wrapper and returned as an error when it is known a priori that the error is handled gracefully, while an exception is not.

In summary, the work performed under this contract has significantly advanced the state-of-the-art in a key area or security and dependability of Win32 systems: software robustness. We've developed and delivered three distinct technologies for analyzing and improving the robustness of Win32 systems under unusual conditions associated either with malicious attack or misbehaving operating system functions.

Dynamic Security Analysis of COTS Applications
Contract No. F30602-97-C-0117

Statement of Work Item 4.1.8.3

Document all technical work accomplished and information generated during the performance of the project. This shall include both positive and negative results, all pertinent observations, and the nature of the problems addressed. The details of all technical work performed under this contract will be documented to permit full understanding of the methods and procedures used in creating the CVA prototype.

1. Introduction

This report documents all technical work accomplished and information generated during the performance of the project F30602-97-C-0117. Section 2 of this report, Subprojects, is divided into three sections that represent each of the primary areas that we have investigated. Each section contains rationale, experimental results, and conclusions for the work performed in that particular area. Section 3 is the Project Summary, and serves to tie together all of the research that we have performed under this contract. Section 4 is a cross-reference that serves to tie together this report and the requirements outlined in the Statement of Work that was specified in the contract. Section 5 contains a list of papers and presentations that have resulted from the work conducted on this project.

2. Subprojects

This section of the report details the three subprojects that make up the bulk of the work performed under this contract. Each subproject addresses a particular aspect of Component Vulnerability Analysis that is essential to solving research problems in this field. The first section, RIDDLE – The Random and Intelligent Data Design Library Environment, explores technologies that are important for the automated testing and security analysis of Commercial Off-The-Shelf (COTS) software components. RIDDLE introduces the development of the grammar generator, the creation of data generators, and the use of a flexible test framework that supports automated testing of COTS software components. Results of using RIDDLE to test software libraries, command-line utilities, and network daemons are presented.

The next section, NetHose – The Network Stack Testing Utility, discusses a prototype tool that has been developed based on the same underlying concepts that came out of the work performed on RIDDLE. This utility addresses the increasingly large concern over Denial of Service (DoS) attacks that have become ubiquitous on the internet. NetHose provides a way to test the robustness of an operating system's network stack when faced with a deluge of valid and anomalous network traffic.

The final section, The Failure Simulation Tool (FST), describes the development and use of a prototype that is used to examine the robustness of Windows NT applications under anomalous environmental conditions. The FST accomplishes this by providing an interface that allows the user to intercept function calls made from an application to any of its Dynamic Link Libraries. An intercepted function call can be caused to fail, simulating failures at the operating system level.

2.1 RIDDLE – the Random and Intelligent Data Design Library Environment

2.1.1 Motivation

An increasingly large number of mission critical applications are relying on the robustness of Commercial Off The Shelf (COTS) software. The military, for one, uses commercially available architectures as the basis for 90% of its systems[1]. Many commercial products are not fully prepared for use in high assurance situations. The testing practices that ordinary commercial products undergo are not thorough enough to guarantee reliability, yet many of these products are being incorporated in critical systems.

High assurance applications require software components that can function correctly even when faced with improper usage or stressful environmental conditions. The degree of tolerance to such situations is referred to as a component's robustness. Most commercial products are not targeted for high assurance applications. These products, which include most desktop applications and operating systems, have not been extensively tested for use in mission critical applications. Despite this fact, many of these products are used as essential components of critical systems.

Given the use of COTS software components in critical systems, it is important that the robustness of these components be evaluated and improved. Studies, including Fuzz [2,3] and Ballista[4], have examined using automated testing techniques to identify robustness failures[5,6]. Automated testing has the advantage of being low-cost and efficient, however its effectiveness depends largely on the data that is used as test input. The input to a component under test will determine which robustness failures (if any) will be discovered, and which will remain hidden. It is therefore essential that high assurance applications be tested with the most effective data possible.

This study examines two different approaches to generating data to be used for automated robustness testing. The two approaches differ in terms of the type of data that is generated, and in the amount of time and effort required to develop the data generation routines. The first type of data generation that is discussed is called generic data generation, and the second is called intelligent data generation. We will compare and contrast both the preparation needed to perform each type of data generation, and the testing results that each yield.

2.1.2. Related Work

Two research projects have independently defined the prior art in assessing system software robustness: Fuzz[2] and Ballista[4]. Both of these research projects have studied the robustness of Unix system software. Fuzz, a University of Wisconsin research project, studied the robustness of Unix system utilities. Ballista, a Carnegie Mellon University research project, studied the robustness of different Unix operating systems when handling exceptional conditions. The methodologies and results from these studies are briefly summarized here to establish the prior art in robustness testing.

2.1.2.1 Fuzz

One of the first noted research studies on the robustness of software was performed by a group out of the University of Wisconsin[2]. In 1990, the group published a study of the reliability of standard Unix utility program[2]. Using a random black-box testing tool called Fuzz, the group found that 25-33% of standard Unix utilities crashed or hung when tested using Fuzz. Five years later, the group repeated and extended the study of Unix utilities using the same basic techniques. The 1995 study found that in spite of advances in software, the failure rate of the systems they tested was still between 18 and 23%[3].

The study also noted differences in the failure rate between commercially developed software versus freely-distributed software such as GNU and Linux. Nine different operating system platforms were tested. Seven out of nine were commercial, while the other two were free software distributions. If one expected higher reliability out of commercial software development processes, then one would be in for a surprise in the results from the Fuzz study. The failure rates of system utilities on commercial versions of Unix ranged from 15-43%while the failure rates of GNU utilities were only 6%.

Though the results from Fuzz analysis were quite revealing, the methodology employed by Fuzz is appealingly simple. Fuzz merely subjects a program to random input streams. The criteria for failure is very coarse, too. The program is considered to fail if it dumps a core file or if it hangs. After submitting a program to random input, Fuzz checks for the presence of a core file or a hung process. If a core file is detected, a ``crash" entry is recorded in a log file. In this fashion, the group was able to study the robustness of Unix utilities to unexpected input.

The causes of crashes were investigated by Fuzz researchers analyzing source code provided by the commercial vendors in addition to the source code available through freely distributed software. Errors that programmers made include pointer/array errors, using dangerous input functions, errors in signed characters, and checking for the end of file when reading input. For example, incrementing the pointer past the end of an array is a common error made by many programmers. Also, the use of dangerous input functions such as the gets() C function can result in program crashes. More insidious manipulation of dangerous input functions can permit ``stack smashing" attacks that allow the execution of arbitrary program code embedded in user input. Another example of a programmer error is assuming that the end-of-file character will always immediately follow a newline character. User input may not necessarily follow this format. Though the Fuzz study did not investigate the vulnerability of programs to buffer overrun attacks, some of the gaps in robustness as measured by the Fuzz study may be exploitable in this manner for security violations.

2.1.2.2 Ballista

Ballista is a research project out of Carnegie Mellon University that is attempting to harden COTS software by analyzing its robustness gaps. Ballista automatically tests operating system software using combinations of both valid and invalid input. By determining where gaps in robustness exist, one goal of the Ballista project is to automatically generate software ``wrappers" to filter dangerous inputs before reaching vulnerable COTS operating system (OS) software.

A robustness gap is defined as the failure of the OS to handle exceptional conditions[4]. Because real-world software is often rife with bugs that can generate unexpected or exception conditions, the goal of Ballista research is to assess the robustness of commercial OSs to handle exception conditions that may be generated by application software.

Unlike the Fuzz research, Ballista focused on assessing the robustness of operating system calls made frequently from desktop software. Empirical results from Ballista research found that read(), write(), open(),close(),fstat(),stat(), and select() were most often called[4]. Rather than generating inputs to the application software that made these system calls, the Ballista research generated test harnesses for these system calls that allowed generation of both valid and invalid input.

Based on the results from testing, a robustness gap severity scale was formulated. The scale categorized failures into the following categories: Crash, Restart, Abort, Silent, and Hindering (CRASH). A failure is defined by the error or success return code, abnormal terminations, or loss of program control. The categorization of failures is more fine-grained than the Fuzz research that categorized failures as either crashes or hangs.

The Ballista robustness testing methodology was applied to five different commercial Unixes: Mach, HP-UX, QNX, LynxOS, and FTX OS that are often used in high-availability, and some-times real-time systems. The results from testing each of the commercial OSs are categorized by the CRASH severity scale and a comparison of the OSs are found in [4].

In summary, the Ballista research has been able to demonstrate robustness gaps in several commercial OSs that are used in mission-critical systems by employing black-box testing. These robustness gaps, in turn, can be used by software developers to improve the software. On the other hand, failing improvement in the software, software crackers may attempt to exploit vulnerabilities in the OS.

The research on Unix system software presented in this section serves as the basis for the robustness testing of the NT software system described herein. The goal of the work presented here is to assess the robustness of application software and system utilities that are commonly used on the NT operating system. By first identifying potential robustness gaps, this work will pave the road to isolating potential vulnerabilities in the Windows NT system.

2.1.3. Input Data Generation

Both the Fuzz project and the Ballista project use automatically generated test data to perform automated robustness testing. The development of the data generators used by the researchers working on the Ballista project clearly required more time than did the development of the data generators used by researchers on the Fuzz project. This is because the Ballista team required a different data generator for each parameter type that they encountered, while the Fuzz team needed only one data generator for all of their experimentation. The data used for command line testing in the Fuzz project consisted simply of randomly generated strings of characters. These randomly generated strings were used to test all of the UNIX utilities, regardless of what the utility expected as its command line argument(s). Each utility, therefore, was treated in a generic manner, and only one data generator was needed. We refer to test data that is not dependent on the specific component being tested as generic data.

The Ballista team took a different approach to data generation. They tested UNIX operating system function calls, and generated function arguments based on the type declared in the function's specification. This approach required that a new data generator be written for each new type that is encountered in a function's specification. Although the number of elements in the set of data generators needed to test a group of functions is less than or equal to the number of functions, this may still require a large number of data generators. We refer to the practice of generating data that is specific to the component currently under test as intelligent data generation.

2.1.3.1 Generic Data

The generation of generic test data is not dependent on the software component being tested. During generic testing, the same test data generator is used to test all components. This concept can be made clearer through an example. When testing command line utilities, generic data consists of randomly generated strings. There are three attributes that can be altered during generic command line utility testing. They are string length, character set, and the number of strings passed as parameters. The same data generators are used to test each command line utility. A utility that expects a file name as a parameter will be tested the same way as a utility that expects the name of a printer as an argument. The test data that the data generator produces is independent of the utility being tested.

2.1.3.2 Intelligent Data

Intelligent test data differs from generic test data because it is tailored specifically to the component under test. The example above can be extended to show the differences between generic and intelligent data. Assume that the current command line utility being tested takes two parameters: a printer name, and a file name. This would require the use of two intelligent data generators (one for generating printer names, the other for generating file names). The intelligent file name generator will produce strings that correspond to existing files. Additionally it will produce other strings that test known boundary conditions associated with file names. For example, on Windows NT there is a limit of 255 characters as the length of a file name. The intelligent data generator will be designed to produce strings that explore this boundary condition. Furthermore, the generator might produce strings that correspond to files with different attributes (read only, system, or hidden), or even directory names. The intelligent printer name generator would produce input data that explores similar aspects of a printer name.

The purpose of using intelligent data generators is to take advantage of our knowledge of what type of input the component under test is expecting. We use this knowledge to produce data that we believe will exercise the component in ways that generic data cannot. Intelligent testing involves combining the use of intelligent data generators with the use of generic data generators. The reason that tests that combine intelligent data with generic data will exercise more of a component's functionality is because the component may be able to screen out tests that use purely generic data. This can be explained by continuing the example of the command line utility that takes a printer name and a file name as its parameters. If the first thing that this utility did was to exit immediately if the specified printer did not exist, then testing with generic data would never cause the utility to execute any further. This would hide any potential flaws that might be found through continued execution of the utility.

2.1.4. The Experiment

In this experiment, we perform robustness testing of Windows NT software components. The two types of components that we test are command line utilities, and Win32 API functions. Both types of components are tested using both generic and intelligent testing techniques.

2.1.4.1 Component Robustness

The IEEE Standard Glossary of Software Engineering Terminology defines robustness as "The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions." (IEEE Std 610.12.1990) Applying this definition of robustness to the two classes of components that we are testing allows us to make two claims.

1. Neither an application, nor a function, should hang, crash, or disrupt the system unless this is a specified behavior.

2. A function that throws an exception that is not documented as being capable of throwing is committing a non-robust action.

The first statement is a fairly straightforward application of the definition of robustness. The second statement requires some more explanation. Exceptions are messages used within a program to indicate that an event outside of the normal flow of execution has occurred. Programmers often make use of exceptions to perform error-handling routines. The danger of using exceptions arises when they are not properly handled. If a function throws an exception, and the application does not catch this exception, then the application will crash. In order to catch an exception, a programmer must put exception-handling code around areas that he or she knows could throw an exception. This will only be done if the programmer knows that it is possible that a function can throw an exception. Because uncaught exceptions are dangerous, it is important that a function only throws exceptions that are documented.

A function that throws an exception when it is not specified that it can throw an exception is committing a non-robust action. The function does not necessarily contain a bug, but it is not performing as robustly as it should. Robustness failures like this can easily lead to non-robust applications.

2.1.4.2 Test Framework

To perform our automated robustness testing we began by developing a simple test framework (Figure 1). The framework consists of four important components: the configuration file, the execution manager, the test child, and the data generation library.

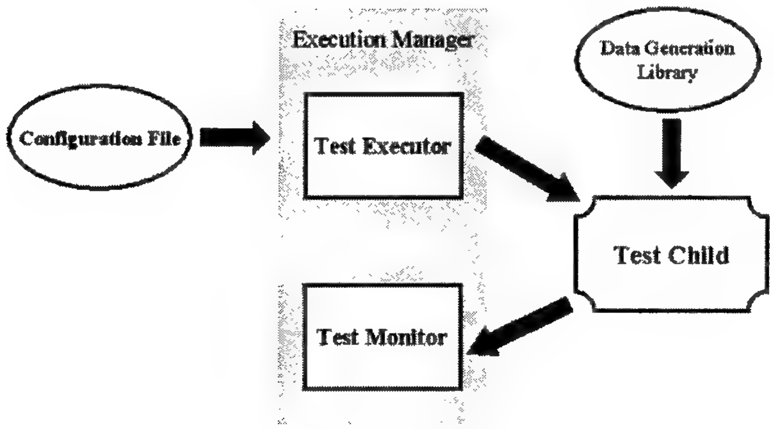


Figure 1: Testing Framework

The configuration file specifies what is being tested, and where the test data will come from. It is a flat text file that is read in one line at a time. Each line includes the name of the component to be tested, and the names of the data generators that should be used to supply the input for each parameter. Each parameter that is required by the component under test is specified individually. This is an example of what a line of the configuration file might look like during intelligent testing. In this example, the utility "print" expects the name of a printer followed by the name of a file.

```
print $PRINTER $FILENAME
```

Here is what a line from the generic testing configuration file might look like:

```
print $GENERIC $GENERIC
```

The data generation library contains all of the routines needed for generating both generic and intelligent data (these are called data generators). Each data generator generates a fixed number of pieces of data. The number of data elements that a data generator will produce can be returned by the data generator if it is queried. The data element that a data generator returns can be controlled by the parameters that are passed to it.

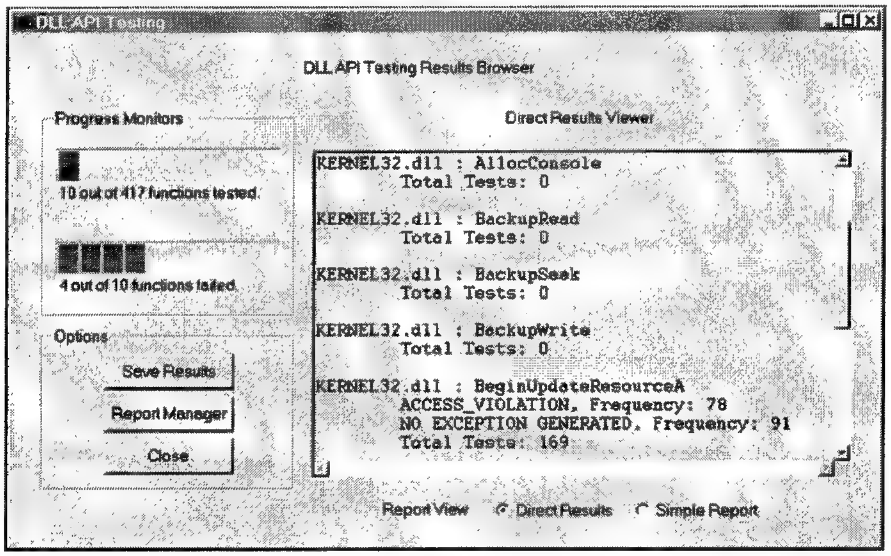
The test child is a process that is executed as an individual test. In the case of the command line utilities, the utility itself constitutes the test child. When testing the Win32 API functions, however, the test child is a special process that will perform one execution of the function under test. This allows each run of a function test to begin in a newly created address space. This reduces the chance that a buildup of system state will affect a test.

The execution manager is the heart of the framework. It is responsible for reading the configuration file, executing a test child, and monitoring the results of the test. After reading a line from the configuration file, the execution manager uses functions in the data generation library to determine how many tests will be run for a component. This number represents all possible combinations of the data produced by the specified data generators. For example, the line from the intelligent testing configuration file mentioned above specifies one file name generator, and one printer name generator. If the $FILENAME data generator produces 10 different values, and the $PRINTER data generator produces 5 values, then the execution manager would know that it has to run 50 (10 x 5) test cases. The execution manager then prepares the test child so that it will execute the correct test. Finally the execution manager executes the test child.

The test monitor is the part of the execution manager that gathers and analyzes the results of an individual test case. The test monitor is able to determine the conditions under which the test child has terminated. Some possible ends to a test case include the test child exiting normally, the test child not exiting (hanging), the test child exiting due to an uncaught exception (program crash), and the test child exiting due to a system crash. In the event of a system crash, after restarting the computer the testing framework is able to continue testing at the point that it left off. The results that the test monitor gathers are used to produce a report that details any robustness failures that were detected during testing.

This framework enables us to configure a set of tests, and then execute them and gather the results automatically. The results are stored as a report that can easily be compared to other reports that the utility has generated.

The complete functionality of the test framework is most easily managed through its Graphical User Interface. The GUI can be used to configure and automatically execute a set of test cases, as well as to produce results reports. Status bars, including those that can be seen in Figure 2, give the tester an idea of how many tests have been run, and how many are remaining. The results of the testing are computed on the fly, and can be viewed in the results window even before all tests have completed. The reports manager, shown in Figure 3 (next page), is used to analyze the results of more than one experiment. It enables the user to select any number of previously generated reports and combine them into a comprehensive report suitable for experimental results comparisons.



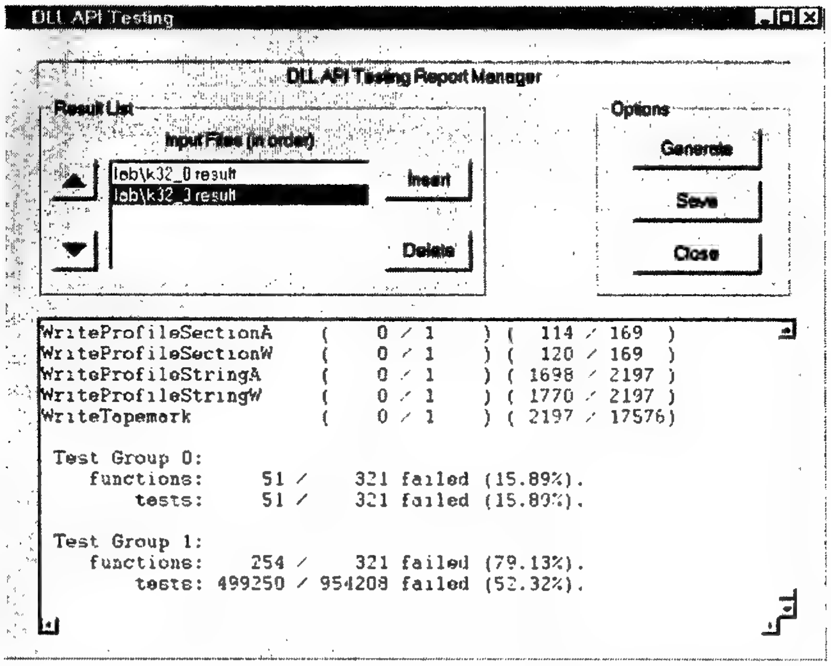
10 out of 417 functions tested.

4 out of 10 functions failed.

```
KERNEL32.dll : AllocConsole
        Total Tests: 0

KERNEL32.dll : BackupRead
        Total Tests: 0

KERNEL32.dll : BackupSeek
        Total Tests: 0

KERNEL32.dll : BackupWrite
        Total Tests: 0

KERNEL32.dll : BeginUpdateResourceA
        ACCESS_VIOLATION, Frequency: 78
        NO EXCEPTION GENERATED, Frequency: 91
        Total Tests: 169
```

Report View [✓ Direct Results] Simple Report

Figure 2: Win32 API Function Testing GUI – Test configuration screen



lob\k32_0 result
lob\k32_3 result

```
WriteProfileSectionA      (     0 / 1      ) (  114 / 169  )
WriteProfileSectionW      (     0 / 1      ) (  120 / 169  )
WriteProfileStringA       (     0 / 1      ) ( 1698 / 2197 )
WriteProfileStringW       (     0 / 1      ) ( 1770 / 2197 )
WriteTopemark             (     0 / 1      ) ( 2197 / 17576)

 Test Group 0:
    functions:      51 /    321 failed (15.89%).
        tests:      51 /    321 failed (15.89%).

 Test Group 1:
    functions:     254 /    321 failed (79.13%).
        tests: 499250 / 954208 failed (52.32%).
```

Figure 3: Win32 API Function Testing GUI – Report manager screen

2.1.4.3 Win32 API Function Testing

The Win32 API is a set of functions that is standard across the Windows NT, Windows 95/98, Win32s, and Windows CE platforms (although not all functions are fully implemented on each of these platforms). These functions are located in Dynamic Link Libraries (DLLs), and represent a programmer's interface to the Windows operating system. For this experiment, we chose to concentrate on three of the most important Windows DLLs: USER32.DLL, KERNEL32.DLL, and GDI32.DLL. The USER32 DLL contains functions for performing user-interface tasks such as window creation and message sending, KERNEL32 consists of functions for managing memory, processes, and threads, and GDI32 contains functions for drawing graphical images and displaying text[7].

2.1.4.3.1 Generic Win32 API Testing

The generic data generators that we used for testing the Win32 API functions were all integer based. This was done because all possible types can be represented through integers. For example, the char * type (a pointer to an array of characters) is simply an integer value that tells where in memory the beginning of the character array is located. The type float is a 32 bit value (just like an integer), and differs only in its interpretation by an application. Since the premise behind generic data generation is that there is no distinction made between argument types, the generic data generator used during the Win32 API testing generates only integers.

The generic testing that we performed on the Win32 API was done in three stages (referred to as Generic 0, Generic 1, and Generic 2). These stages are distinguished by the sets of integers that we used. Each set of integers is a superset of the previous set. The first set of integers that we used consisted only of { 0 }. The second consisted of { -1, 0, 1 }, and the third contained { -2^{31}, -2^{15}, -1, 0, 1, $2^{15} - 1$, $2^{31} - 1$ }. A test consisted of executing a function using all combinations of the numbers in these sets. For example, during the first stage of testing we called all of the functions and passed the value zero as each of the required parameters (resulting in only one test case per function). The second stage of testing consisted of running 3^X test cases, where x is the number of parameters that the function expects. The final stage of testing required 7^X test cases. Due to the time intensive nature of the testing that we are conducting, we limited our experiment to test only functions that contained four or fewer parameters (a maximum of $7^4 = 2401$ tests per function during generic testing).

2.1.4.3.2 Intelligent Win32 API Testing

Intelligent testing of the Win32 API involved the development of over 40 distinct data generators. Each data generator produced data that is specific to a particular parameter type. One data generator was often capable of producing multiple pieces of data related to the data type for which it was written. Furthermore, each intelligent data generator also produced all of the data items output by the third generic data generator. An example of an intelligent data generator is the data generator that produces character strings. In addition to the data produced by the third generic data generator, this data generator produces a number of valid strings of various lengths and the null string. Other examples of Win32 API intelligent data generators are those that produce handles to files, handles to various system objects (i.e., module handles), and certain data structures.

2.1.4.3.3 Win32 API Testing Results

The results of both the generic and intelligent Win32 API experimentation are summarized in Figure 4. The bars represent the percentage of functions that demonstrated robustness failures. The robustness failures that are charted are almost entirely due to exceptions that the functions are throwing. Any one of these exceptions could cause an application to crash if they are not caught. The most common exception that we found (representative of an estimated 99% of all exceptions) is an ACCESS_VIOLATION. This is a type of memory exception that is caused by a program referencing a portion of memory that it has not been allocated. An example of this would be trying to write to a null pointer. "Access violations" frequently occur when an incorrect value is passed to a function that expects some sort of pointer. The number of functions that throw access violations when they are passed all zeros underscores this point. Keep in mind that the undocumented throwing of an exception does not necessarily indicate that a function contains a bug, however this is often not the most robust course of action that the function could take.

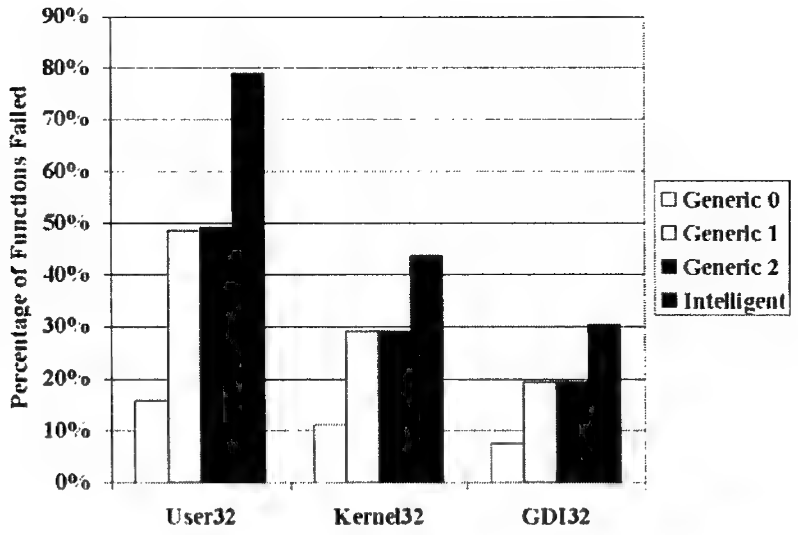


Figure 4: Win32 Function Testing Results

There were 321 USER32 functions tested, 307 KERNEL32 functions tested, and 231 GDI32 functions tested. In the analysis that we present here, we focus on the number of functions in each DLL that demonstrated robustness failures. There are a couple of reasons that we chose to concentrate on the number of functions that had failures, not the number of failures per function. One reason for this is that a function only has to suffer a robustness failure once to potentially harm a mission critical system. If it can be shown that a function is capable of failing in at least one circumstance, then the robustness of this function is called into question.

Another reason for focusing on the percentage of functions that failed, not the percentage of failures per function, is that the number of tests that are run for each function is subjective. This subjectivity arises from the development of the data generators. There is no practical way to write a data generator (generic or intelligent) that will produce an exhaustive set of tests. Look, for example, at the intelligent data generator that produces strings. We use this generator to produce a number of strings of varying lengths. There is, however, a near infinite number of string lengths and character patterns that we could produce. Instead of attempting to exhaustively generate all of these possibilities (an intractable task), we instead select a small sampling of strings that we hope will test a function in different ways.

The way that this subjectivity could affect our data gathering is if we examine the number or percentage of failures per function. We would not even be able to give an accurate percentage of failures on a per function basis. If function X fails 50% of the time when tested with the data used during the third round of generic testing, we could easily add or remove data values to alter this percentage. What is most important to us is not the number of times that we can cause a function to fail, but whether or not a function failed during our testing.

Each progressive level of testing, from Generic 0 to intelligent, is a superset of the previous testing level. Generic testing accounted for over 60% of the exceptions found in each of the three DLLs tested. Notice that the percentage of functions that fails rises sharply between the first and second rounds of generic testing, and then again between the third round of generic testing and the round of intelligent testing. These results indicate two things to us. First, they show that despite its simplicity, generic testing is a worthwhile activity. Second, the results indicate that intelligent testing is a more comprehensive, and thus necessary part of automated robustness testing.

The process of setting up the generic tests that we conducted on the Win32 API functions was a fairly inexpensive task. The generic data generators that we used were simple to design and implement. Additionally, it appears that further expanding the sets of integers used during generic testing will not bring many new results. As the set of integers was changed from three elements to seven elements, the number of additional functions that failed was zero for KERNEL32 and GDI32, and only two for USER32. The generic data generation approach to automated testing could certainly be valuable to a tester that simply wanted to take a first step towards evaluating the robustness of a set of functions.

The significant increase between the number of functions found to exhibit robustness failures during generic testing, and the number of functions found to fail during intelligent testing underscores the importance of intelligent software testing. This data supports the claim that as the level of intelligence that is used during testing increases, so does the number of problems discovered. For critical systems, this indicates that a significant amount of time and effort needs to be put into software testing.

As a final note, we found a number of serious operating system robustness failures during the testing of GDI32 and KERNEL32. Each of these DLLs contains a few functions that were capable of crashing the operating system. All of these operating system crashes occurred during intelligent testing. These OS crashes are significant because they represent robustness failures that could not be trapped by an application in any way. They are dangerous because they could either occur accidentally during normal system use, or could be caused intentionally by a malicious user (as in a Denial Of Service attack).

2.1.4.4 Win32 Command Line Utility Testing

For the second part to this experiment, we performed automated robustness testing of a number of Windows NT command line utilities. The primary difference between this experiment and the testing of the Win32 API functions is that we are now testing complete applications, not operating system functions. The utilities that we tested consisted of both a group of programs that are native to the Windows NT operating system, and a group of programs that were written by Cygnus for use on the Windows NT platform. Many of the command line utilities that we tested are analogous to the UNIX utilities tested by the researchers on the Fuzz project.

2.1.4.4.1 Generic Command Line Utility Testing

The generic data generators that we used for testing the command line utilities were all string based. The parameters that command line utilities accept are always read from the command line as strings. Therefore, if a command line utility expects an integer as a parameter, it reads these parameters as a string (e.g., "10") and then converts it to the type that it requires.

There is another fundamental difference between the Win32 API function testing and the command line utility testing that we need to address. All of the functions that we tested accepted a fixed number of parameters. Many command line utilities will accept a varying number of parameters. Parameters are delimited by blank spaces. Instead of only judging whether or not a command line utility exhibited any robustness failures, we chose to distinguish between test cases that involved different numbers of parameters. Due to resource considerations, we limited the number of parameters that we would test to four.

The following is an example of what the configuration file for the command line utility "comp" looks like:

```
comp $GENERIC
comp $GENERIC $GENERIC
comp $GENERIC $GENERIC $GENERIC
comp $GENERIC $GENERIC $GENERIC $GENERIC
```

We refer to each of these lines as a template. Each utility is tested using 1, 2, 3, and 4 parameters, for a total of four templates. The generic data generator for the command line utility testing produced strings of varying lengths and character sets. The character sets that were used included alphanumeric, printable ASCII, and all ASCII except the null character. The null character (ASCII value 0) was avoided because it could be interpreted as the end of a line. Additionally, none of these strings contained spaces, so they should not be misinterpreted as representing more than one parameter.

2.1.4.4.2 Intelligent Command Line Utility Testing

The intelligent data that was used as parameters to the command line utilities was based on syntactic and semantic information gathered from documentation for each utility. Although the data generators produce only strings, the strings that they produce have semantic meaning. For example, an intelligent data generator that produces file names chooses names of files that actually exist. There exist other data generators that produce strings corresponding to integers, directory names, printer names, etc. In addition to using these intelligent pieces of data, intelligent testing involved running test cases that combined intelligent data with generic data.

2.1.4.4.3 Command Line Utility Testing Results

Tables 1 and 2 summarize the results of the command line utility testing. The data represents the number of templates that exhibited robustness failures (out of a possible four). The robustness failures charted here are due to the application abnormally terminating due to an uncaught exception that was thrown within the application. As stated earlier, an exception, such as those examined in the Win32 API function testing portion of this experiment, can cause an application to crash if it is not handled properly.

Robustness Failures Discovered in Microsoft NT Utilities

	Generic	Intelligent
findstr	3 / 4	4 / 4
xcopy	3 / 4	4 / 4
expand	2 / 4	3 / 4
comp	0 / 4	1 / 4
ftp	0 / 4	1 / 4
ping	0 / 4	0 / 4

Table 1: Results from native Windows NT command line utility testing

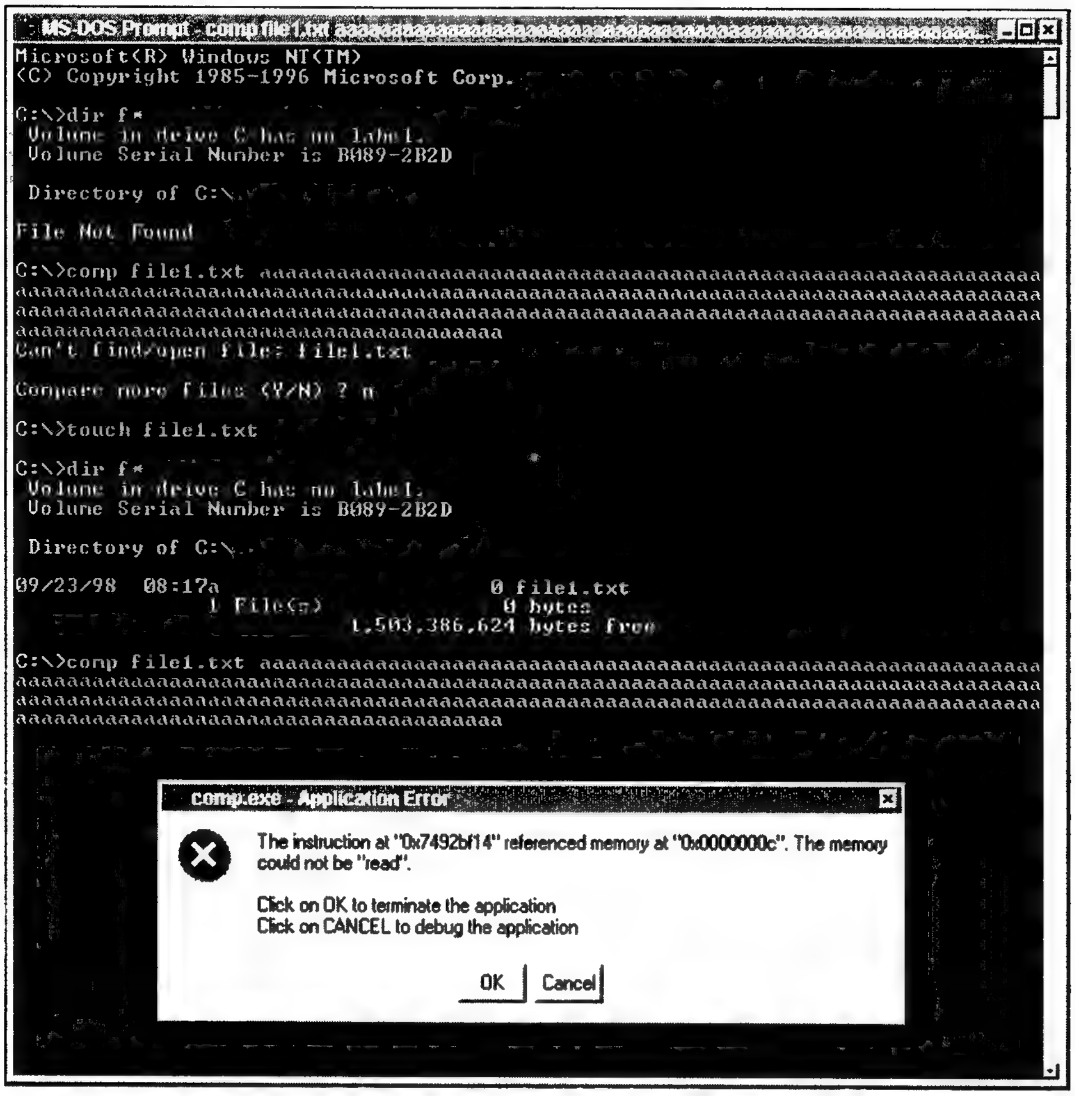
Robustness Failures Discovered in Cygnus GNU Win32 Utilities

	Generic	Intelligent
diff	2 / 4	3 / 4
gunzip	2 / 4	2 / 4
ls	2 / 4	3 / 4
cp	2 / 4	3 / 4
od	2 / 4	3 / 4
grep	2 / 4	2 / 4

Table 2: Results from testing Cygnus Windows NT command line utilities

The experimental results that we have gathered during the command line utility testing appear to support the conclusions that we came to after analyzing the Win32 API function testing. Simple generic testing was able to uncover robustness failures in a significant number of the utilities that we tested (9 out of 12). Intelligent testing was able to uncover two additional utilities that contained robustness failures.

Intelligent testing also proved to be better than generic testing at discovering more robustness failures for each utility. In all but three of the twelve utilities tested, intelligent testing resulted in more failed templates than generic testing. Only one of the tested utilities did not demonstrate any robustness failures.


MS-DOS Prompt - comp file1.txt aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...

```
Microsoft(R) Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.

C:\>dir f*
 Volume in drive C has no label.
 Volume Serial Number is B089-2B2D

 Directory of C:\

File Not Found

C:\>comp file1.txt aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Can't find/open file: file1.txt

Compare more files (Y/N) ? n

C:\>touch file1.txt

C:\>dir f*
 Volume in drive C has no label.
 Volume Serial Number is B089-2B2D

 Directory of C:\

09/23/98  08:17a                    0 file1.txt
               1 File(s)             0 bytes
                          1,503,386,624 bytes free

C:\>comp file1.txt aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

comp.exe - Application Error

The instruction at "0x7492bf14" referenced memory at "0x0000000c". The memory could not be "read".

Click on OK to terminate the application
Click on CANCEL to debug the application

Figure 5: Screen capture of a robustness failure discovered in the native Windows NT command line utility comp.

Figure 5 contains a screen capture of a robustness failure that we discovered in the native Windows NT utility comp. This particular failure occurs only when the first parameter is a valid file name and the second parameter is a buffer of approximately 250 characters. This is an example of a robustness failure that could not be detected using only generic testing because it only appears when the first parameter is the name of an existing file.

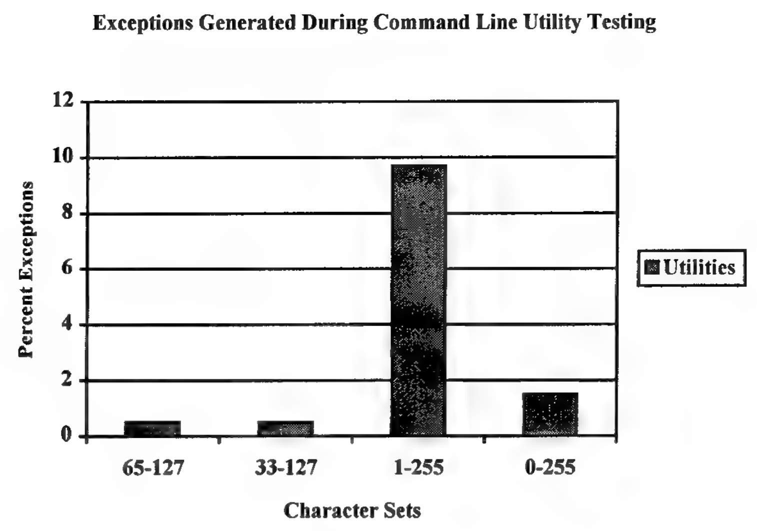


Figure 6.

A further breakdown of these experimental results shows that the vast majority of the exceptions occurred when the character set being used for string generation was in the range (1,255) (excluding the NULL character). This is shown in Figure 6, which depicts the percent of test cases that resulted in exceptions. This pattern is most likely due to the program's interpretation of special characters. The number of exceptions decreases dramatically when the character set is altered to include the null character, or when it consists only of printable characters in the range (33,127). The null character may be interpreted as the termination character of a string, effectively limiting the length of the input. This would explain why there are fewer unhandled exceptions when this character is used in light of the correlation between length and exceptions (see Figure 7). Another possibility is that if the null character is interpreted as either the end of a string or the end of the parameter list, then the parameters may no longer constitute a valid use of the application and the utility may immediately reject the test case.

Clearly, these command-line utilities are most vulnerable to input that is sampled from the character set range (1,255). This set includes every printable and non-printable character except for the NULL character. Even very long length input in the alphabetical and printable set resulted in few exceptions. Instead, it is the combination of very long length with nearly the entire range of the character set (including non-printable characters) that resulted in the most unhandled exceptions. The most significant trend in the data collected from these tests is the increase in unhandled exceptions as the length of string increases as illustrated in Figure 7.

The graph of the exception ratios show that as the length of input is increased from 8 to 4096 bytes, the number of exceptions rises dramatically indicating a failure to handle anomalous input within proper input grammar. Significantly fewer exceptions occurred when the length of the string used was either 8 or 250 characters. Because the exception that occurred most often was a memory access violation, the cause is most likely an over-written buffer that placed an illegal pointer on the program stack. In other words, the instruction pointer that was overwritten with the long input probably points to a region of the memory that is inaccessible for the program, or it may point to data that is not a valid instruction opcode. This result points to potential vulnerabilities in these utilities to buffer overrun attacks. Buffer overrun attacks are one of the most significant security-related flaws that are most often exploited in practice[8,9]. The Fuzz study also pointed out the relative vulnerability of programs to unconstrained input[3]. However, the assertion that these programs are vulnerable to buffer overrun attacks has not been investigated in this study.

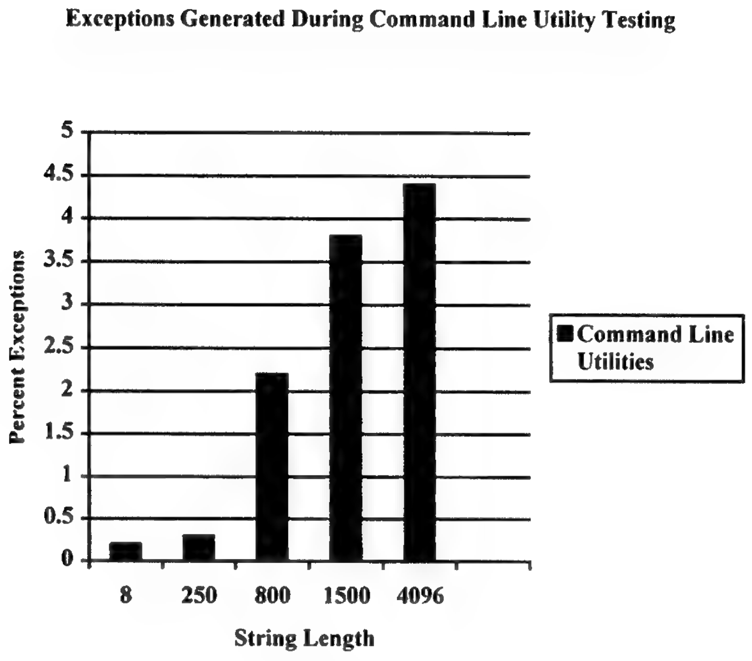


Figure 7.

2.1.4.5 The Grammar Generator

The grammar generator takes a different approach to producing valid and anomalous data than the techniques presented above. Specifically, the grammar generator addresses the fact that the subject being tested may not take a fixed number of inputs. The grammar generator requires that the user is able to specify the inputs to the test subject, and then it can produce streams of data that conform to these specifications.

2.1.4.5.1 Grammar Generator Theory

The ability to generate intelligent input is essential for the type of stress testing necessary for robustness assessment. The simplest form of testing involves generating random streams of data that are used by the program being tested; this was done in the Fuzz project. While random input generation can test the ability of a program to handle non-conforming input, it typically will not exercise much of a program's functionality. Testing applications with syntactically correct data will result in more thorough testing of that application than testing with purely random data. For example, many applications that take command line arguments will immediately terminate if they do not receive the correct number of parameters, or if they receive an invalid flag. In this situation, random testing will not test any further into the program than this initial check.

On the other hand, syntactically correct arguments (or input parameters) will result in more of the application being tested. In order to exercise more of a program's functionality and to test more of the function's response to anomalous input, we have supplemented RIDDLE with a grammar-based input generation component. With the creation of a grammar-based input generation component, RIDDLE can test software with syntactically correct data that contains unexpected, anomalous input. The anomalous input itself will be generated through function calls to the data generation library.

The grammar generator takes a grammar specification as input, and produces random, yet syntactically correct, strings of data by employing functions from the data generation library. The goal of this approach is to produce input that is closer to being valid than that produced through the template method that has been discussed above.

The data generation library can be used to generate data with a variety of levels of intelligence. When testing an application that takes a file name as a parameter, the data generation library can be used to produce a number of substitutions for this parameter. In the case of a file name, the data generation library can produce the name of an existing file, a valid file name that doesn't exist, an invalid file name, the name of a file with specific permissions set, an extremely long file name, or otherwise. Each of these possibilities results in a different test case that may exercise the application being tested in a new way.

2.1.4.5.2 Architecture

The grammar specification is defined in two parts. The first part is a definition of the grammar written in a format similar to Backus-Naur Form (BNF). The second part is a file that contains definitions of all of the tokens used in the grammar. RIDDLE begins by parsing the grammar definition and checking that it is syntactically correct. Next, RIDDLE begins the process of generating data based on the grammar that it has read. The data that RIDDLE generates relies on the terminal definitions that have been supplied in the token definition file and the functions called from the data generation library.

For each program being tested, the grammar definition must be created. The definition declares the format for the input that is syntactically correct for exercising the program under test. Each production, or rule, consists of a left-hand side and a right-hand side, separated by a colon. The left-hand side identifies a single non-terminal. The right-hand side of the production identifies a set of non-terminals and tokens that the non-terminal on the left-hand side can reduce to. A single non-terminal could have a number of choices of reductions. In this case, each reduction is separated by the symbol ``|''. Tokens are productions that reduce only to a single terminal (they are therefore only one step away from being terminals themselves). In RIDDLE's case, terminals always reduce to a string. The terminal that a token reduces to is specified separately from the grammar definition in the token definition file.

The following simple grammar is equivalent to the regular expression (a)*b, or any number of a's followed by a single b (i.e.}, b, ab, aab, etc).

Grammar Definition

```
START:      B
      |     A     START ;
```

Token Definitions

```
A:      "a"
B:      "b"
```

RIDDLE always begins with the first production rule that is given. All possible syntactically correct sentences must begin from this production. In this example, the production that defines the non-terminal START is the starting production. When reducing the non-terminal START, the data generator has a choice of picking either the production rule that results in token B (in the example above), or the production rule that results in the token A followed by the non-terminal START. The data generation component uses a repeatable random number generator to choose between the production choices that it faces. Additionally, RIDDLE provides a means of weighting the choice of production rules. This probability is specified by adding a weight to the end of a production. The START production could be written as:

```
START:     B          1
      |    A START    3
```

This grammar definition indicates that the chance of non-terminal START reducing to A START is 3 times more likely then it reducing to B. Or in other words, there is a 75% chance that it will reduce to A START and a 25% chance that it will reduce to B.

When the grammar-based data generation component is called upon to produce syntactically correct data it begins with the starting production. It then chooses productions at random (taking into account the probabilities that were added) until all of the non-terminals have been reduced to tokens. For example, the grammar above might produce the string of tokens AAAAB. The final step of the process is to reduce the tokens to their values. In our example, A reduces to the string ``a" and B reduces to string ``b", giving us the string ``aaaab". The data that is produced will be syntactically correct with respect to the grammar that was used to create it.

RIDDLE allows the user to specify that a token reduces to either a string literal, or a function that returns a string literal. In the previous example, token B reduced to the string literal ``b". The user could specify that token B reduce to the function call

RandomFileName(). This function could return a string that would be used in place of the token B (e.g., the choices of file names that were mentioned earlier). This is how RIDDLE is able to generate data that will serve as anomalous program input. The functions that can be used in the reduction of a token make up the data generation library. This library will contain functions that reduce to numerous user-specified strings that can be used for testing purposes.

RIDDLE is designed to test two types of applications: those that take input from the command line, and those that take streams of data as input. The former class of applications includes many commonly used operating system utilities. Examples of such Unix utilities include the cp, ls, man, and ps commands. Windows NT examples include mode, tree, subst, and format. Applications that rely on streams of data include Web servers (httpd), ftp servers, ftp clients, lpr, and grep.

A simplified example of testing the UNIX utility cp can be demonstrated. This grammar definition only accounts for a small subset of the command's functionality, but it is useful for illustrative purposes.

Grammar Definition

```
START:     SP     LOW_F      START
      |    SP     LOW_P      START
      |    SP     FILE_NAME FILE_NAME ;
```

Token Definition

```
LOW_F:      "-f"
LOW_P:      "-p"
FILE_NAME: GenerateFileName()
SP:         " "
```

This grammar specification will provide for the production of an input string that consists of any number of -f's and -p's followed by two strings produced by the GenerateFileName() function. The input strings that are produced could then be used in test cases. The following are some test cases that RIDDLE might produce:

```
cp -p oaisud aoisudf
cp -p -f -p -p <existing file> <existing directory>
cp -f -p <open file> <extremely long buffer of characters>
```

These test cases are syntactically correct usages of the cp command, combined with anomalous data. If the anomalous data that is supplied by the data generation library results in undesirable application failure, then a weakness in the robustness of the application has been detected.

2.1.4.5.3 Command-Line Utility Testing with the Grammar Generator

As discussed, the grammar generator is useful for producing input that can be used to test components that accept an arbitrary number of inputs. We used the grammar generator to produce input strings to test command-line utilities and network daemons. The first step to this process was to produce grammar definitions and token definitions for the components to be tested. The grammar definitions that were used to test command-line utilities were developed from the usage documentation for these utilities. The grammar definitions for the network daemons that we tested were available as BNF definitions from developer documentation that is publicly available.

We ran tests on the same set of Microsoft and GNU utilities that we tested using the template approach (section 2.1.4.4). The number of test cases that can be produced for each utility using the grammar generator is completely arbitrary. The length of the input strings that are generated is also arbitrary. Because the nature of this testing is so subjective, we simply chose to run a number of test cases that fell within the time constraints that we had set for this testing.

MS Utilities	Exceptions Generated
Findstr	Yes
Xcopy	Yes
Expand	Yes
Comp	Yes
ftp	Yes
Ping	No

Table 3.

GNU Utilities	Exceptions Generated
Diff	Yes
Gunzip	Yes
Ls	Yes
Cp	Yes
Od	Yes
Grep	Yes

Table 4.

Tables 3 and 4 show the results of the tests that we ran on the command-line utilities. This table indicates whether or not the utility being tested threw any exceptions during the test period. It would not be scientifically valid to calculate the percent of failures during these tests because the number of test cases is arbitrary. The results of the tests that were produced using the grammar generator match those gathered through the template testing discussed in section 2.1.4.4. The same utilities that displayed failures during the intelligent template testing failed during testing with the grammar generator. This indicates that the template testing that we performed was able to achieve comparable results to the grammar generator testing, with significantly less work required. To strengthen this claim, we would need to considerably extend the range of utilities that were tested using each approach. Due to the amount of effort required for this experimentation, we have not performed such validation at this time.

2.1.4.5.4 Network Daemon Testing with the Grammar Generator

The network daemons that we tested were web servers and ftp servers. The first step towards testing these applications was to modify RIDDLE so that it could transmit data across a network. The watchdog process was removed, because the application being tested was no longer local to the machine running RIDDLE. Instead of putting a watchdog process on the remote machine, RIDDLE looked for application failures by periodically verifying that the daemon being tested was still responding correctly. With these modifications in place, RIDDLE was now able to perform the remote testing of network daemons.

For the web server testing, we worked from a BNF grammar that was obtained from a public distribution. The grammar files that we used for web server testing were significantly more complicated than those used for the testing of command-line utilities. We then wrote a token definition file that contained the data generators that we were interested in using.

The web servers that were tested are Microsoft's Internet Information Server 3.0 (IIS) and the APACHE Web Server. Although we subjected these servers to extensive testing, we were unable to find any robustness problems with either server. This is likely due to years of bug fixes and application hardening that these applications have gone through. A robustness flaw in a web server would be a major problem for any company that relies on the availability of their web server. When such flaws are found, they are often exploited as Denial of Service attacks by malicious users in the computer community. Developers of web servers must remain diligent in fixing any robustness flaws that are discovered in their product.

The grammar definition files that were used for the testing of the file transfer protocol servers (FTP servers) were also based on publicly available specifications. The FTP servers that were tested are Microsoft's IIS 3.0 and Solaris' `ftpd`. The testing of these servers did not result in any robustness failures. Once again, these are essential applications that are generally accessible to anyone that is connected to the internet. Years of exposure to network attacks have resulted in robust applications that are good at handling anomalous network traffic.

2.1.5. Conclusions

The experimental results of both the Win32 API function testing and the Windows NT command line utility testing demonstrate the usefulness of performing automated robustness tests with generic data, as well as the importance of using intelligent robustness testing techniques. Despite its simplicity, generic testing has proven to provide valuable results in both halves of this experiment. Automated generic testing is an inexpensive yet useful testing technique.

Intelligent testing uncovered more robustness failures for both the automated Win32 API function robustness testing, and the automated Windows NT command line utility testing. These results verify that more intelligent testing techniques uncover more robustness failures. Furthermore, intelligent testing uncovers robustness failures that could never be discovered using only generic testing (as described in the example of the comp utility).

The desire to build fault tolerant computer systems using commercial software necessitates better testing of software components. This involves the testing of both existing components, and of the system as a whole. The experiment conducted indicates that automated robustness testing using generic testing techniques can yield impressive results, but that mission critical applications will require significantly more intelligent testing.

2.2 NetHose – A Tool for Testing the Robustness of the Network Stack

2.2.1. Introduction

The networking of computers is largely responsible for the revolutionary effect computers have had in the workplace and in our lives. Before Microsoft Windows, most personal computers running MS-DOS were standalone systems. Today, almost every computer system is networked with other computer systems, with the ultimate network being the Internet. The software that implements the TCP/IP protocol used by the Internet is called the network stack. The network stack software is a critical component in the operating system (OS), and by extension, to the national information infrastructure (NII). It is the portion of the OS that processes network packets before and after network services receive and send them. Since the internet is becoming more homogenous than ever, a flaw in the network stack of one of the dominant platforms can leave a large portion of the NII vulnerable to attack. Thus, in order to assure survivability of the critical NII, we must ensure that the network stack software is robust to anomalous conditions or malicious attack.

In this section of the report, we discuss our approach and describe a tool that tests the robustness of the network stack software to anomalous conditions and malicious attack. The approach is based on our previous work in testing the robustness of OS software by using combinations of valid and invalid input, as described in section 2.1. The key distinction, however, is that this section describes the application of these techniques towards a tool for generating anomalous network packets in order to test the robustness of OS networking software.

The importance of this work is in developing structured approaches to testing critical portions of the software that composes the NII. In addition, a structured approach to testing enables scientific study of the robustness of the platform. One of the key problems in developing a survivable NII is the robustness of a given platform to denial of service attacks. Vulnerability to denial of service attacks is the Achilles heel of today's Internet. There are an increasingly large number of Denial of Service attacks that are publicly available on the Internet, and OS vendors are scrambling to release patches preventing known attacks. Fundamental problems in the design of network protocols can leave all network implementations vulnerable. However, more pervasive than design flaws, are software implementation flaws, or bugs. Flaws in the networking software can leave a platform vulnerable to misuse of network services or to malicious attack. The goal of the approach developed here is to test the network stack for robustness to these types of unusual conditions in order to identify flaws in the platform that leave it vulnerable. The testing approach and tool described here applies equally to any software platform that is networked, however, results from applying the tool to the Win32 platform are presented here.

2.2.2. Approach

One possible approach to examining the network stack for weaknesses would be to perform a thorough analysis of the operating system's source code. This is a software engineering task that the vendor should perform prior to releasing the system. We are interested in determining the robustness of network stacks for COTS software. With the notable exception of Linux, most OS vendors choose not to make the source code to their operating systems available. As a result, we must use an approach that is source code independent.

Another technique that could be used for network stack analysis is a black-box approach. The traditional black-box approach to testing involves generating inputs with respect to functional specifications (not source code) and analyzing the output. This technique has the advantage of not requiring any operating system source code, and therefore being platform independent. The biggest issue with this type of testing will be the generation of test data, and the analysis of the results.

The approach taken here is a black-box testing approach, but rather than generating inputs to perform functional testing, we generalize combinations of valid and invalid inputs to test robustness. The IEEE Standard Glossary of Software Engineering Terminology defines robustness as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" (IEEE Std 610.12.1990).

Central to this approach is the design of data generator units that can continually produce test data according to our robustness testing criteria. The NetHose testing tool provides a framework that makes use of data generators to produce an automated network stack testing tool.

It is important to note here that we are not testing the OS to determine how robust it is under heavy load conditions. This approach is known as stress testing or load testing, and several commercially available tools exist for this purpose. We will concern ourselves only with testing how robust the network stack is to anomalous input.

2.2.3. Network Stack Overview

To develop our technique for testing the network stack, it is first necessary to garner a better understanding of the inner workings of the network stack. Simply put, the network stack is the portion of the operating system that makes inter-computer communication possible.

The implementation of the TCP/IP protocol suite in the operating systems is often referred to as the network stack. This is because the protocol headers are pushed onto a message during sending and then popped off during receiving (demultiplexing). Conceptually, most network stacks are loosely based on the 7 layer OSI model, but can easily be depicted as having 4 layers (Figure 8).

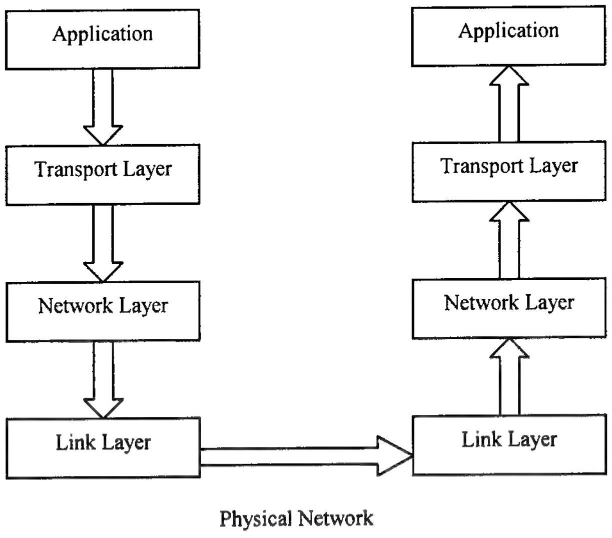


Figure 8. Path of Data through the Network Stack

When an application wishes to send a message across a network, a stack of 3 headers is placed onto that message by the OS. First a transport layer header describes how the data should appear to be transmitted from application to application. Then a network layer header maps out how the data should travel across the network. Finally, a link layer header describes the physical details of interfacing the cable. TCP and UDP are the two most common transport level protocols, and IP is the underlying network layer protocol.

2.2.3.1 Protocol Headers

16-bit source port number			16-bit destination port number
32-bit sequence number			
32-bit acknowledgement number			
4-bit hdr length	6-bit reserved	6-bit flags	16-bit window size
16-bit checksum			16-bit urgent pointer

Figure 9: The TCP Header

The TCP protocol provides for a reliable and connection-oriented byte stream service. By storing select information in the protocol headers (Figure 9), TCP provides for such properties as data integrity and port delivery. For instance, it ensures the integrity of the data by computing an end-to-end checksum and then storing it in the header. For packet delivery, however, TCP relies on a network level protocol, namely IP.

16-bit source port number	16-bit destination port number
16-bit datagram length	16-bit checksum

Figure 10: The UDP Header

The UDP protocol, also a transport layer protocol, provides an unreliable datagram service. It has only a few special properties: port delivery, and data integrity. To provide these services, UDP stores information in the header (Figure 10). It stores, for instance, a destination port number and an end-to-end checksum in the header, however it does not provide for re-transmission of lost packets.

4-bit version	4-bit hdr length	8-bit type of service	16-bit total length	
16-bit identification field			3-bit flags	13-bit fragment offset
8-bit time to live		8-bit protocol	16-bit header checksum	
32-bit source address				
32-bit destination address				

Figure 11: The IP Header

The IP protocol is the predominant network protocol. It provides a best-effort packet delivery service that is used by all the transport protocols in the TCP/IP suite (see Figure 11 for a depiction of the IP header). However, there is no guarantee that an IP packet will arrive at its destination, or on arrival, be intact. These qualities are provided by higher level protocols[10].

2.2.3.2 IP Fragmentation

The IP protocol makes provisions for the fragmentation and re-assembly of IP packets that are deemed too large. IP fragmentation can occur at the source machine or anywhere along the route to the destination. It is a process whereby the data segment of an IP packet is broken into multiple pieces, each of which receives an IP header. The IP header of each fragment contains a fragmentation offset that the destination machine uses to put the pieces back together. IP fragmentation is completely transparent to both the source and destination transport layer. For instance, consider a single UDP datagram consisting of 16 header bytes and 32 data bytes. The contents of the UDP datagram are usually placed into a single IP packet and sent. With IP fragmentation, however, a UDP packet may be split into several pieces during its journey, and then the pieces are reassembled on arrival at their destination.

2.2.3.3 Berkeley Raw Sockets

Berkeley Sockets are the traditional programming interface to the network stack. The sockets API allows for the creation of sockets that can be connected, read, and written by an application. During the creation of a socket, the application can specify what sort of transport service it would like. Sockets, then, represent the interface between the application and the transport layer. Each layer below the application layer is then filled in by the operating system.

Should a programmer need to over-ride the operating system and fill in these layers himself, he can do this by making use of raw sockets. For each layer that the programmer wishes to emulate, additional space is created at the top of the message buffer. This space can then be filled with the correct (or incorrect) values that correspond to what the OS fills in. For instance, the OS places the source IP address in the IP header. By custom crafting the IP field, however, the programmer can place any IP address he desires into that field. This is often referred to as IP spoofing, and it is one potentially malicious use of raw sockets. Another, more common, use for raw sockets is to simulate transport protocols not supported by regular sockets. For instance, a programmer wishing to send an ICMP echo request can fill in the ICMP and IP protocol headers by hand in order to create the traditional "ping" effect. The link layer, which most often entails the addition of an Ethernet header, is added by the OS, and cannot be manipulated with raw sockets.

2.2.4. Data Generators

As mentioned earlier, the most significant problem of black-box testing is the generation of test data. Unlike traditional black-box testing approaches, NetHose generates data in order to test the bounds of the software under analysis. The approach uses combinations of valid and invalid inputs to test the network stack. The packets that NetHose creates will be used to test the network stack's ability to handle anomalous data.

NetHose uses data generators to produce data to fill each of the fields in the header of a packet. The network stack processes the packet headers, not the body of the message. As a result we are interested in generating anomalous packet header fields. Each field has its own data generator that is capable of producing values specific to that field. The values that a data generator produces are a mix of valid and anomalous data. For example, the data generator that produces the value to be used in the "header length" field will be able to produce the correct value, but is also able to produce values that are incorrect. The incorrect values will attempt to test boundary conditions and special circumstances. For example, incorrect values that the "header length" field might produce include values that are slightly longer or slightly shorter than the actual value, the value zero, and the maximum value allowed.

NetHose attempts to exploit the relationships that exist between certain header fields. To capture these relationships, NetHose allows data generators to be built through the use of other data generators. For example, if field A is supposed to be less than field B, then a data generator for field A might be built that first calls the data generator for field B, and then generates its own data. The values that data generator A produces are based on the relationship that exists between fields A and B. These values test the boundaries of this relationship by producing data that violates the relationship in a variety of ways.

The tests that NetHose executes consist of using data generators together in different combinations. Ideally we would execute every combination of each data generator, however running this many tests is currently not feasible. What we have done is to group together related header fields. We then execute tests that explore all combinations of these groups of fields.

2.2.5. Test Framework

The primary component in NetHose is the testing module, which runs on the testing machine. This module is responsible for reading the configuration file, building the appropriate packets, and sending these packets. NetHose also makes use of a small module that runs on the machine that is being tested. The function of this module is simply to return a "ping" to the machine that is performing the testing. This is used to determine whether the machine being tested is still responding.

The testing module begins by reading from a configuration file. This file contains information that describes the tests that are going to be run. First, it contains information to describe which protocols are under test. Then it describes which fields in the transport header need to be perturbed. For instance, the message size field and the checksum fields might be selected for testing. The file then specifies how many fragments the transport layer message should be broken into. Finally, for each fragment, fields in the network layer header are selected for testing. Fields such as IP version and IP type of service can be selected. NetHose parses all of this information and stores it internally.

Next, NetHose begins to construct the packets that will be used in the test. The data generators create the data that fills the packets. NetHose combines the data generators in the manner specified in the configuration file.

Once the packet has been created, it is sent over the network via the raw sockets interface. After sending each sequence of packets, NetHose sends a message to the machine that is being tested. If the machine is still functioning properly, it will reply to this message. When NetHose has detected that a machine is no longer responding, it will pause the testing process. At this point NetHose also makes a record of the disruption in service. These disruptions are further examined in a more controlled environment. Testing resumes when NetHose is once again able to communicate with the target machine, usually after the machine has been manually restarted.

2.2.6. Methodology

It is hardly feasible to test all possible combinations of bits in even a relatively small header such as the UDP header. Admittedly, such a test set would constitute a very thorough measurement of the robustness of a UDP implementation. However, the UDP header consists of 4 fields that are 16 bits in size, and 2^{64} is a prohibitively large number of packets to send across the network. Furthermore, an overwhelming majority of these combinations will fail the same initial sanity checks. For instance, they will fail the checksum. If, however, the number of tests per field is limited to a few carefully crafted values, the resulting set of packets may serve to approximate the useful set of tests. To be feasible then, black-box testing of the network stack requires a carefully limited set of tests for each field.

For protocols that are not stateless, such as TCP, and to a lesser degree, IP, it is not necessarily sufficient to test all combinations of a single packet. To fully test the IP fragment re-assembly portions of an IP implementation, it is necessary to construct a test set that includes multiple IP packets. If you consider that each IP header consists of 12 fields, and that each field can be tested by 4 carefully crafted values, it still becomes necessary to send 4^{12} packets across the network for each fragment. Conducting a test that contains two fragments would require 4^{24} packets. This is clearly infeasible for testing multiple IP fragments, and it is similarly infeasible for testing multiple packets in a TCP session. Thus, black-box testing of the network stack must involve the careful selection of fields under test.

When testing a single packet, we ran a single large test set that we hoped would uncover any errors in the stateless aspects of the UDP and IP protocols. Any errors could be isolated using progressively smaller test sets. To test the IP re-assembly algorithms, it became important to selectively narrow the number of fields under test. This whittling down of the test set was accomplished in a few different ways: certain fields of only moderate relevance were tested very little and other fields of little or no relevance are left unperturbed. If, for instance, we wished to test IP re-assembly with two packets, we might only perturb the IP Type Of Service Flags in one of the packets. Furthermore, we might refrain from perturbing the IP Version field at all. These methods of reducing the test set were employed at the discretion of the tester.

2.2.7. Testing Setup

The testing that we conducted was performed on three different platforms: Windows NT Service Pack 3, Windows NT Service Pack 4, and Windows 95 OEM Service Release 2. Several different machines were used during testing, including a laptop, a 486, a dual Pentium, and several other high-end workstations. The pace of packets being sent to the machines under test was intentionally slow enough that performance differences in machines should not affect the outcome of the tests.

2.2.8. Description of Robustness Weaknesses Found

Three types of robustness failures were observed during the testing: kernel exceptions, hard freezes, and system slowdown. The slowdown that we observed occurred only when testing the Windows 95 operating system. After a large number of tests, it became impossible to load and execute any applications on the Windows 95 system. The system returned to normal after being rebooted.

A kernel exception (colloquially known as the Blue Screen of Death) appeared during several tests of NT SP3, and Windows 95. Under Windows 95, the user has the option of either rebooting or continuing execution in a corrupted environment. On NT the system must be rebooted.

The test computer sometimes completely froze during a batch of tests. This was the most common catastrophic result. After freezing, the test computer no longer responded to the mouse, keyboard, or network. The screen would remain frozen in place until the computer was manually reset. This type of robustness failure was found on both NT SP3 and Windows 95.

2.2.9. Testing Results

An exhaustive testing of a single UDP/IP packet required the transmission of nearly 100000 packets. This involved varying all 4 UDP fields and 7 IP fields in one giant set of tests. The only robustness failure uncovered during this set of tests was an instance of slowdown on Windows 95. Thereafter, we set about testing multiple IP fragments in a large number of smaller test sets. The fields varied during this series of test sets were chosen heuristically in order to limit the number of tests. This series of tests sometimes involved as few as 100 packets or as many as 10,000 packets. The fragmented packet tests were much more successful than the single packet tests at uncovering failures, and they uncovered several dozen robustness weaknesses in both Windows NT SP3 and Windows 95 OSR 2. The failures uncovered included both blue screens and hard freezes.

2.2.10. Conclusions

The common thread between all of the test sets that caused failures is that they all involved perturbation of the IP fragmentation offset field. This would lead us to conclude that for Windows 95 and Windows NT Service Pack 3, insufficient checking was done on the fragment offset field. Because none of these same tests uncovered any robustness failures in Windows NT Service Pack 4, we can assume that this set of robustness failures has been since corrected, and that Windows NT SP4 is robust to fragmentation attacks. This makes sense because one of the purposes of SP4 was to protect against fragmentation attacks.

The most significant errors in the UDP and IP implementations lay in the IP. The fact that an IP implementation requires some internal state for fragment re-assembly makes the IP protocol a good target for NetHose style testing. It is possible then, that implementations of a protocol such as TCP will contain similar errors because TCP maintains state for a variety of purposes.

2.3. The Failure Simulation Tool

2.3.1 Introduction

Commercial off-the-shelf (COTS) software is being used increasingly in developing mission-critical systems. For the purposes of our research, COTS software is any software for which source code is not available. However, in general, off-the-shelf software is any software that is not developed in house. The time and expense of developing software in house has spurred developers of critical systems in the transportation, medical devices, and nuclear industries to adopt COTS software in the development of their critical systems. More recently, the Windows 32-bit (Win32) platform, which includes Windows 95/NT/2000/CE operating systems, is being used in mission critical applications.

For example, the U.S. Navy requires that its ships migrate to Windows NT workstations and servers under the Information Technology in the 21st century (IT-21) directive[11]. While modernizing the fleet's technology base is appropriate, the risks of migrating to new platforms are great, particularly in mission-critical systems. One widely-publicized early casualty of this directive involved the USS Yorktown, a U.S. Navy Aegis missile cruiser. The cruiser suffered a significant software problem in the Windows NT systems that control the ship's propulsion system. An application crash resulting from an unhandled exception reportedly caused the ship's propulsion system to fail, requiring the boat to be towed back to the Norfolk Naval Base shipyard[12].

We believe the migration of critical systems to COTS software and to the Windows platform will continue. However, in spite of the headlong rush to adopt COTS software, there exists a dearth of research, technology, and tools for determining the impact of failures of third-party COTS software on the dependability of the system[13]. That is, the system integrators or maintainers of critical systems have little support to make engineering decisions on what kind of impact the failure of a third party software component will have on their systems, let alone know how to harden their systems for robustness to failures from third party off-the-shelf software. The reality is, no one develops any system, soup to nuts, from custom-built software. Instead, an organization will run its software (or even purchase the application software) on commercial operating systems and use third-party software components to build their applications. In mission-critical systems, it is imperative that the impact of the failure of these third-party components on the application software be known in advance in order to harden the software for robustness.

In this research topic we develop an approach and technology for artificially forcing exceptions and error conditions from third-party COTS software when invoked by the software application under study. The goal in developing this technology is to support testing of critical applications under unusual, but known, failure conditions in an application's environment. In particular, we simulate the failure of operating system functions; however, the approach and tool can be used to simulate the failure of other third party software such as imported libraries and software development kits.

We have focused on the Win32 platform because we believe this to be the platform to which most critical applications are migrating, and it is the platform for which the least amount of research on dependability assessment has been performed, in spite of its growing adoption. While the general approach developed here is platform independent, the technology we have built and the implementation details of the approach are specific to the Win32 platform.

The approach is briefly summarized here, then developed in Section 2.3.4. Because we are working in the domain of COTS software, we do not assume access to program source code; instead, we work with executable program binaries. The approach is to employ fault injection functions in the interface between the software application under study and the operating system (or third party) software functions the application uses. The fault injection functions simulate the failure of these resources, specifically by throwing exceptions or returning error values from the third-party functions. The simulated failures are not arbitrary, but rather based on actual observed failures from OS functions determined in our previous study of the Windows NT platform (see [6]), or based on specifications of exceptions and error values that are produced by the function being used. In addition, the approach does not work on models of systems, but on actual system software itself. Therefore, we are not simulating in the traditional sense, but rather forcing certain conditions to occur via fault injection testing that would otherwise be very difficult to obtain in traditional testing of the application. The analysis studies the behavior of the software application under these stressful conditions and poses the questions: is the application robust to this type of OS function failure? does the application crash when presented with this exception or error value? or does the application handle the anomalous condition gracefully?

In the remainder of this section, we present some background in robustness testing research, provide motivation for why handling errors and exceptions is critical, then develop the methodology and tool for testing COTS software under these types of stressful conditions.

2.3.2 Background

Robustness testing is now being recognized within the dependability research community as an important part of dependability assessment. To date, robustness testing has focused on different variants of Unix software. In section 2.1.2 we discussed the work performed by B.P. Miller[2,3] and P. Koopman[4,14].

In this study, we are concerned with testing the robustness of application software --- specifically mission-critical applications --- that run on the Win32 platform. Unlike nominal testing approaches (see [15,16,17,18,19]) that focus on function feature testing, we are concerned with testing the software application under stressful conditions. Robustness testing aims to show the ability, or conversely, the inability, of a program to continue to operate under anomalous input conditions. More formally, the IEEE Standard Glossary of Software Engineering Terminology states that robustness is ``the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions".

Testing an application's robustness to unusual or stressful conditions is generally the domain of fault injection analysis. To date, fault injection analysis of software has generally required access to source code for instrumentation and mutation (see [20] for an overview of software fault injection). In addition, fault injection analysis to date has been performed on Unix-based systems. We seek to develop technologies that will work on COTS-based systems and for the Win32 platform.

To define the problem domain of this work better: an application is robust when it does not hang, crash, or disrupt the system in the presence of anomalous or invalid inputs, or stressful environmental conditions. Applications can be vulnerable to non-robust behavior from the operating system. For example, if an OS function throws an unspecified exception, then an application will have little chance of recovering, unless it is designed specifically to handle unspecified exceptions. As a result, application robustness is compromised by non-robust OS behavior.

In our previous studies of the Windows NT platform, we analyzed the robustness of Windows NT OS functions to unexpected or anomalous inputs[5,6]. We developed test harnesses and test data generators for testing OS functions with combinations of valid and anomalous inputs in three core Dynamically Linked Libraries (DLLs) of the Win32 Application Programming Interface (API): USER32.DLL, KERNEL32.DLL, and GDI32.DLL. Results from these studies show non-robust behavior from a large percentage of tested DLL functions. This information is particularly relevant to application developers that use these functions. That is, unless application developers are building in robustness to handle exceptions thrown by these functions, their applications may crash if they use these functions in unexpected ways.

We know from our testing of the Win32 platform that the OS functions can throw exceptions and return error values when presented with unusual or ill-formed input. In fact, we know exactly which exceptions and error codes a given OS function will return based on function specifications and our previous experimentation. However, even though this anomalous behavior from the OS is possible (as demonstrated), it is actually unusual during the normal course of events. That is, during normal operation, the OS will rarely behave in this way. Using nominal testing approaches to test the application might take an extremely long time (and a great many test cases) before the OS exhibits this kind of behavior. Thus, testing the robustness of the application to stressful environmental conditions is very difficult using nominal testing approaches.

However, using fault injection, we force these unusual conditions from the OS or from other third-party software to occur. Rather than randomly selecting state values to inject, we inject known exception and error conditions. This approach then forces these rare events that can occur to occur. Thus, this approach enables testing of applications to rare, but real failure modes in the system. The approach does not completely address the problem of covering all failure modes (especially unknown ones), but it does exploit the fact that third-party software does fail in known ways, even if infrequently. At a minimum, a mission-critical application must account for these known failure modes from third-party software. However, many applications never do, because software designers are mostly concerned about the application they are developing and assume the rest of the environment works as advertised. The approach and tool developed here tests the validity of that assumption by forcing anomalous conditions to occur.

It is important to note that we are not necessarily identifying program bugs in the application, but rather we are assessing the ability of the application to handle stressful environmental conditions from third-party software. So, this approach is an off-nominal testing approach that is not a substitute for traditional testing and fault removal techniques. In fact, the approach tests the application's error and exception handling mechanisms --- the safety net for any application. If the application does not account for these types of unusual conditions, chances are that it will fail.

2.3.3 Errors and Exceptions

Error and exception handling are critical functions in any application. In fact, error and exception handling make up a significant percentage of the code written in today's applications. However, error and exception handling are rarely tested because: (1) programmers tend to assume well-behaved functionality from the environment, and (2) it is difficult to create these kinds of anomalous behaviors using nominal testing techniques.

In [21], Howell argues that error handling is one of the most crucial, but most often overlooked aspect of critical system design and analysis. For example, Howell cites four examples of the criticality of error handling[21]:

- an analysis of software defects by Hewlett-Packard's Scientific Instruments Division determined that error checking code was the third most frequent cause of defects in their software
- in a case study of a fault-tolerant electronic switching system, it was found that 2 out of 3 system failures were due to problems in the error handling code
- many of the safety-critical failures found in the final checks of the space shuttle avionics system were found to be associated with the exception handling and redundancy management software
- problems with the use of Ada exceptions were a key part of the loss of the first Ariane-5 rocket

Errors and exceptions are often used by vendors of third-party software components to signal when a resource request has failed or when a resource is being improperly used. For example, exceptions may be thrown when invalid parameters are sent to a function call, or when the requesting software does not have the appropriate permission to request the resource.

Error codes are returned by a function call when an error occurs during the execution of the function. For example, if a memory allocation function is unable to allocate memory, it may return an invalid pointer. Error codes are graceful exits from a function that also allow a programmer to debug the source of the problem. Exceptions can be thrown for similar reasons, but more often, exceptions are thrown for more severe problems such as hardware failures or for cases when it is not clear why a resource request failed. For example, an exception returned by a function call may have actually originated from another function that was called by the requested function. If an exception is not handled by one function, it is passed up the function call chain repeatedly until either some function handles the exception or the application crashes.

In using third-party software, the application developer must be aware of what exceptions can be thrown by the third-party function. Third-party functions can be embedded in libraries such as the C run-time library, software development kits, commercial software APIs, or as part of the core operating system. In most cases, the source code to the third-party function is not available, but header files and API specifications are. The function API, header files, or documentation should declare what exceptions, if any, can be thrown by the third-party function (and under what circumstances). If the application developer does not write exception handlers for these specified cases, then the robustness or survivability of the application is placed at risk if an exception is thrown in practice.

2.3.4 Wrapping Win32 COTS Software for Failure Simulation

In order to assess the robustness of COTS-based systems, we instrument the interfaces between the software application and the operating system with a software wrapper. The wrapper simulates the effect of failing system resources, such as memory allocation errors, network failures, file input/output (I/O) problems, as well as the range of exceptions that can be thrown by OS functions when improperly used. The analysis tests the robustness of the application to anomalous and stressful environment conditions. An application is considered robust when it does not hang, crash, or disrupt the system in the presence of anomalous or invalid inputs, or stressful environmental conditions.

From our previous studies of the Windows NT platform, we found a large percentage of OS functions in the three core DLLs of the Win32 API that threw exceptions when presented anomalous input. If these functions are used similarly by an application, then the application must be prepared to handle these exceptions, specified or not. Because testing the application via nominal testing approaches is unlikely to trigger these anomalous OS conditions, we need some alternative method to test the robustness of these applications to OS anomalies without requiring access to source code. To address this shortcoming in the state-of-the-art, we have developed the Failure Simulation Tool (FST) for Windows NT.

The approach that FST employs is to artificially inject an error or exception thrown by an OS function and determine if the application is robust to this type of unusual condition. FST instruments the interface between the application executable and the DLL functions it imports such that all interactions between the application and the operating system can be captured and manipulated.

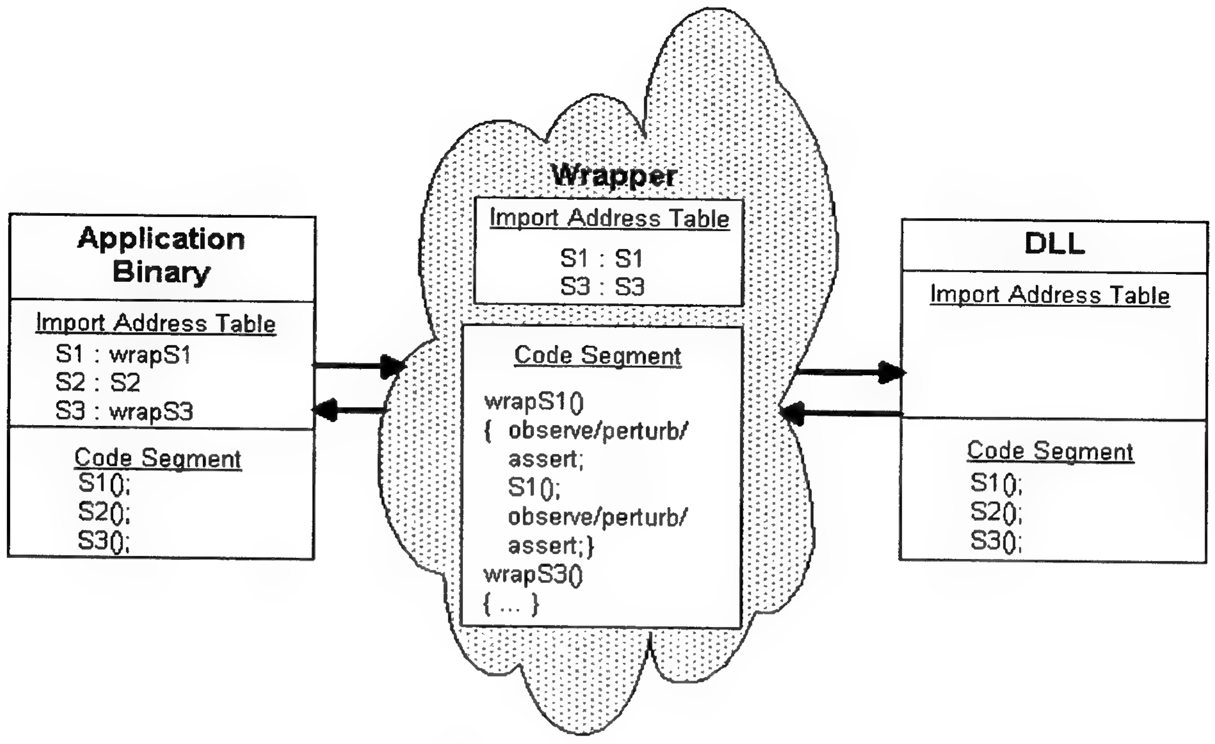


Figure 12: Wrapping Win32 Executable Programs

Figure 12 illustrates how program executables are wrapped. The application's Import Address Table (IAT), which is used to look up the address of imported DLL functions, is modified for functions that are wrapped to point to the wrapper DLL. For instance, in Figure, functions S1 and S3 are wrapped by modifying the IAT of the application. When functions S1 and S3 are called by the application, the wrapper DLL is called instead. The wrapper DLL, in turn, executes, providing the ability to throw an exception or return an error code to the calling application. In addition, as the figure shows, we also have the ability to record usage profiles of function calls and the ability to execute assertions.

There are several ways in which the wrapper can be used. First, the wrapper can be used as a pass-through recorder in which the function call is unchanged, but the function call and its parameters are recorded. This information can be useful in other problem domains such as for performance monitoring and for sandboxing programs. Second, the wrapper can be used to call alternative functions instead of the one requested. This approach can be used to customize COTS software for one's own purposes. For our purposes, we are interested in returning error codes and exceptions for specified function calls. Thus, we develop custom failure functions for each function we are interested in failing.

Three options for failing function calls are: (1) replace calling parameters with invalid parameters that are known to cause exceptions or error codes, (2) calling the function with the original parameters that are passed by the application, then replacing the returned result with an exception or error code, or (3) intercept the function call with the wrapper as before, but rather than calling the requested function, just returning the exception or error code. Option 3 is attractive in its simplicity. However, options (1) and

(2) are attractive for maintaining consistent system state. In some cases, it is desirable to require the requested function to execute with invalid parameters to ensure that side effects from failing function calls are properly executed. Option (1) accounts for this case. Using the information from our previous studies of the Win32 API[6], Option (1) can be implemented by using specifically those input parameters that resulted in OS function exceptions. Alternatively, specifications for which parameters are invalid when using a function (such as one might find in pre-condition assertions) can be used for causing the failure of the function in using Option (1). Option (2) is less rigorous about handling side effects from failing function calls, but will ensure side effects from normal function calls are properly executed. If, however, side effects from calling functions are not a concern, i.e. , if the analyst is strictly concerned about how well the application handles exceptions or error codes returned from a function call, then Option 3 is sufficient.

The FST modifies the executable program's IAT such that the address of imported DLL functions is replaced with the address to our wrapper functions. This modification occurs in memory rather than on disk, so the program is not changed permanently. The wrapper then makes the call to the intended OS function either with the program's data or with erroneous data. On the return from the OS function, the wrapper has the option to return the values unmodified, to return erroneous values, or to throw exceptions. We use this capability to throw exceptions from functions in the OS (which we found to be non-robust in our earlier studies) called by the program under analysis.

After instrumenting all of the relevant Import Address Table entries, the FST performs a search across the code segment of the module for call sites to the wrapped functions. A call site is identified as a call instruction followed by a one-word pointer into the IAT. This call site information is used for two purposes. First, it gives the user a sense of how many call sites there are for a particular function. Second, it allows the FST to tally the number of calls on per site basis instead of on a per function basis.

Finally, the FST tries to match call sites with linked or external debug information. For instance, if the FST has located a call to HeapAlloc at a specific location in memory, it attempts to determine the corresponding line of source. Though debugging information is not necessarily distributed with an application, it can be a great help to a tester on those occasions when it is present. Because of the difficulties involved with juggling the many formats in which debugging information can be distributed, the FST makes makes use of the Windows NT library ImageHlp.DLL which helps to abstract the details of the debug information format.

The FST is also able to intercept dynamic calls to exported functions. The FST provides a special wrapper for the KERNEL32.DLL function GetProcAddress, a function that can be used to retrieve the address of a function at run-time. For functions that the FST would like to wrap, the special GetProcAddress wrapper will return the address of a wrapper function instead of the correct function.

2.3.5 Using the Failure Simulation Tool

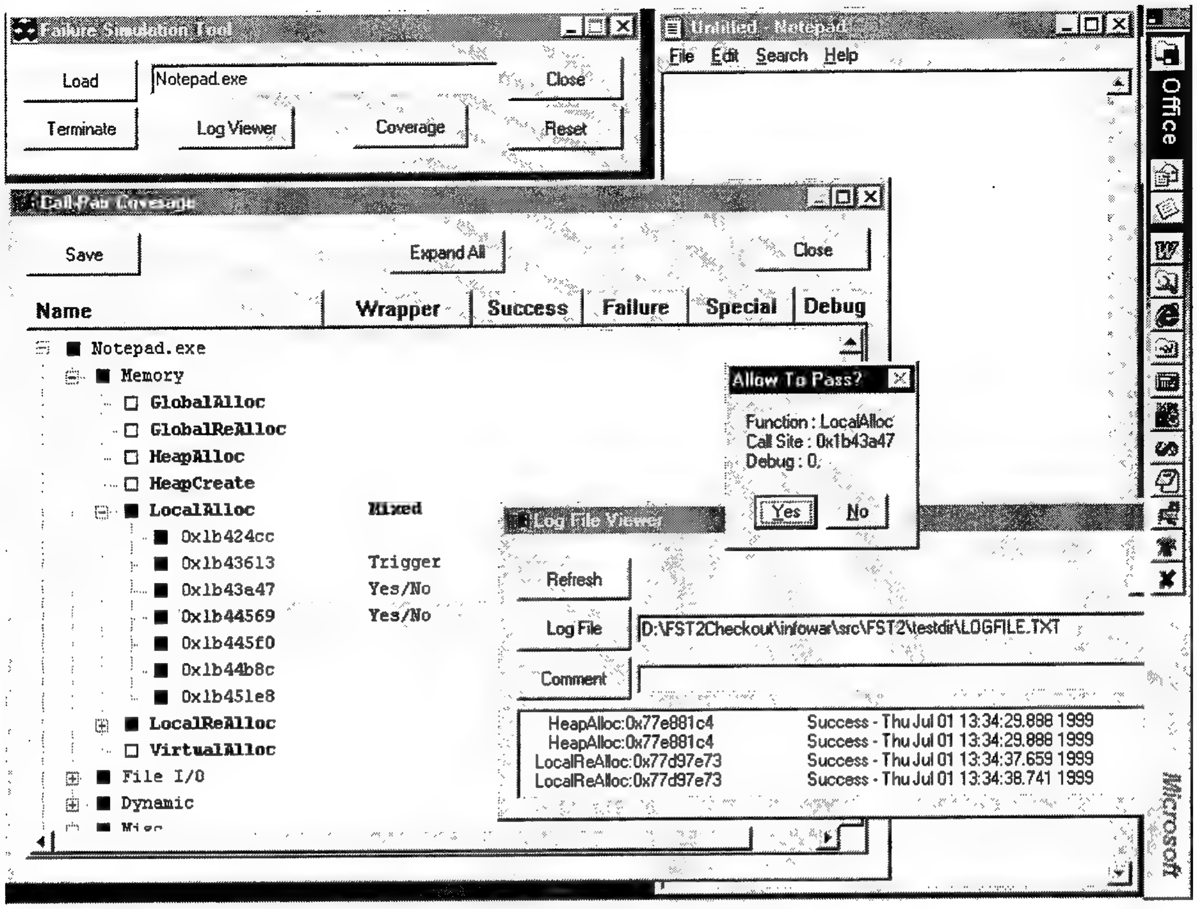
The prototype Failure Simulation Tool provides an interface that helps to simplify the testing of COTS software. There are three primary components to the Failure Simulation Tool: the Graphical User Interface (GUI), the configuration file, and the function wrapper DLLs.

A function wrapper DLL contains the wrappers that will be called in place of the functions that the user chooses to wrap. These wrappers are responsible for simulating the failure of a particular function. The FST includes a variety of wrappers for commonly used functions. Additional wrappers can be developed by a user and used by the Failure Simulation Tool.

The configuration file is used to specify the DLL functions that are going to be wrapped, and the function that is going to be doing the wrapping. The configuration file is also used to break the functions into groups that are displayed by the tree control in the GUI. Here is an example of the configuration file.

```
PAGE: Memory
     KERNEL32:HeapAlloc:WRAPDLL:WrapHeapAlloc
     KERNEL32:LocalAlloc:WRAPDLL:WrapLocalAlloc
```

This configuration file specifies that there should be a group named "Memory," and that there will be two functions in that group. The first function to wrap is HeapAlloc, located in KERNEL32.DLL, and it should be wrapped using the function WrapHeapAlloc, found in WRAPDLL.DLL. The second function is specified in the same manner.



Notepad.exe

D:\FST2Checkout\infowar\src\FST2\testdir\LOGFILE.TXT

Function : LocalAlloc
Call Site : 0x1b43a47
Debug : 0

Name	Wrapper	Success	Failure	Special	Debug
Notepad.exe					
Memory					
GlobalAlloc					
GlobalReAlloc					
HeapAlloc					
HeapCreate					
LocalAlloc	Mixed				
0x1b424cc					
0x1b43613	Trigger				
0x1b43a47	Yes/No				
0x1b44569	Yes/No				
0x1b445f0					
0x1b44b8c					
0x1b451e8					
LocalReAlloc					
VirtualAlloc					
File I/O					
Dynamic					
Misc					

```
HeapAlloc:0x77e881c4        Success - Thu Jul 01 13:34:29.888 1999
HeapAlloc:0x77e881c4        Success - Thu Jul 01 13:34:29.888 1999
LocalReAlloc:0x77d97e73     Success - Thu Jul 01 13:34:37.659 1999
LocalReAlloc:0x77d97e73     Success - Thu Jul 01 13:34:38.741 1999
```

Figure 13: The Failure Simulation Tool GUI

Figure 13 shows the graphic interface to the failure simulation tool that allows selective failing of OS resources. The window in the upper left is the Execution Manager. This window is used to select an application for testing, to execute and terminate that application, and to access the other components of the FST. The largest window in Figure 13 is the interface that allows the user to control which functions are currently being wrapped, and what the behavior of those wrappers should be. The window titled "Log File Viewer" maintains a time-stamped listing of all of the function calls that have been made. The small window in the foreground is an example of a wrapper that allows the user to select whether a function should fail or not on a case-by-case basis. In the upper right of this figure is the application being tested – in this case the Notepad.exe application. For a more detailed description of how to use the Failure Simulation Tool, please see the documentation supplied as part of the System User's Manual.

In its current version, the FST is used to interactively fail system resources during execution. The dynamic binding to functions allows the tool to fail or succeed calls to OS functions on the fly in real time. The FST can be used to wrap any Win32 application (mission critical or not) in order to interactively fail system resources and to determine the impact of this failure. The performance overhead of the wrapping has not been measured, however, no visible degradation in performance of the application software has been observed. The tool is to be used for off-line analysis, however, in order to determine the effect of system failures prior to deployment. Thus, performance overhead in the analysis is not a large concern, unless it were to introduce unacceptable delays in testing.

2.3.6 Conclusions

This section on the Failure Simulation Tool provides an approach and tool for assessing the robustness of Win32 applications in the face of operating system anomalies. Two factors have motivated this work: first, more and more critical systems are being employed on the Win32 platforms such as Windows NT/95/CE/2000; second, the error/exception handling routines of software applications are rarely tested, but form the critical safety net for software applications. In addition, because most COTS software (such as third-party libraries or OS software) rarely provides access to source code, we constrain our approach to analyzing software in executable format.

The Win32 Failure Simulation Tool was developed to allow interactive failing of OS resources during testing. The tool allows the analyst to observe the effect of errors or exceptions returned from the OS on the application under analysis. If the program fails to handle exceptions thrown by an OS function it will usually crash.

In the example of the USS Yorktown, the approach described herein can be used to wrap the ship's propulsion system software in order to assess how robust it is to exceptions thrown by the OS. For instance, when a divide-by-zero exception is thrown, the analysis would show that the propulsion system will crash. This information can then be used to prevent such an error from occuring or to handle the divide-by-zero exception gracefully. The most pressing question now for this smart ship is what other exceptions is the ship's propulsion system and other critical systems non-robust to?

Currently, a limitation of the tool is its coarse-grained ability to monitor the effects of the fault injection on the target application. Our measure for robustness is crude: an application should not hang, crash, or disrupt the system in the presence of the failure conditions we force. However, it is possible that the third party failures we introduce slowly corrupt the program state (including memory and program registers) that remain latent until a later period after the testing has ceased. It is also possible that while the failure does not crash the program it could simply cause the incorrect execution of the program's functions. Neither of these cases is analyzed by our tool. To address the former problem, an extensive testing suite would be necessary to gain confidence that even after the failure was caused, the program remains robust. To address the latter problem, an oracle of correct behavior is necessary and a regression test suite would be required after the failure was forced in order to determine correctness of the program. In both of these cases, our tool would falsely label the program as robust to the failure of the third-party component, when in fact the failure introduced a latent error in the application, or the program's output is corrupted without effecting the execution capability of the program. Hence, the scope of our monitoring is limited to the ability of the program to continue to execute in the presence of third-party failures. It does not have the ability to judge the correctness of the functions computed by the application.

In summary, the testing approach and prototype presented is of value to consumers, integrators, and maintainers of critical systems who require high levels of confidence that the software will behave robustly in the face of anomalous system behavior.

3. Project Summary

In this project, we examined a key property of secure and dependable systems: robustness. Our research focused on developing techniques for analyzing as well as increasing the robustness of software due to unknown and anomalous events. The project was partitioned into three distinct and complementary threads of robustness research: intelligent test case generation, testing of the Win32 network stack for resilience to denial of service, and testing application software for robustness to failing operating system (OS) resources.

The Random and Intelligent Data Design Library Environment (RIDDLE) was developed to support intelligent black-box testing of operating system and command-line utilities to unusual inputs. The work leveraged research in robustness testing of OS utilities previously performed on Unix systems, most notably by Barton Miller's research group at the University of Wisconsin. Robustness testing of OS utilities has previously found vulnerabilities in operating system software to unexpected, random input. Adopting this approach for the Win32 platform, we enhanced it further by intelligently combining valid input with anomalous input in order to unmask flaws that remain hidden to purely random testing. RIDDLE provides a test harness and library for automatically generating test data according to both random (generally invalid) and valid parameters. Our studies have shown empirically the benefit derived from combining intelligent test data generation with random test case generation for the purpose of testing robustness.

Drawing on the experience of RIDDLE, we developed a tool for testing the robustness of the Win32 network stack to anomalous data. The importance of this work is in developing structured approaches to testing critical portions of software that comprises the National Information Infrastructure (NII). Since the network stack of the Win32 platform is a key component of the NII, this approach is among the first to independently and systematically study the robustness of this software under anomalous input conditions. Fundamental problems in the design of network protocols such as TCP/IP can leave all platforms vulnerable. However, flaws, in the networking software can leave a given platform vulnerable to anomalous use of network services, or attack. NetHose is a network stack testing utility that is able to test the network stack of Win32 systems to unexpected data. This type of analysis effectively tests the operating system's ability to handle unusual network packets. The approach uses combinations of valid and invalid data in packet header fields in order to test the robustness of the network stack against unusual packet headers. Our studies revealed three types of robustness failures: kernel exceptions (colloquially known as blue screen of death), hard freezes, and system slowdowns.

The third thread of research in this project was concerned with testing the robustness of Win32 applications under failing OS conditions. From our previous research with RIDDLE, we found that the three core libraries that compose the Win32 system more often than not throw memory access violation exceptions when presented unusual input. Thus, if application developers (particularly for mission-critical applications) do not account for these exceptions that the OS throws, then the application is likely to crash. The Failure Simulation Tool (FST) provides the ability to test Win32 executables (without requiring source code) for robustness to exceptions or errors returned by OS functions. FST instruments the interface between an application and the OS DLL in order to return errors or exceptions from an OS function. This is a far more efficient approach than black-box testing of an application in hopes of generating an OS exception. Experiments showed that Microsoft desktop applications had varying levels of non-robustness to exceptions and errors returned by OS functions. This type of non-robust behavior is typically expected from desktop applications. However, non-robustness to errors or exceptions returned from OS functions is typically not acceptable in a mission-critical application, such as a ship propulsion system. Thus, FST provides the ability to test mission-critical software for robustness to failing OS functions. In the final stage of this work, we used the instrumentation layer to provide protective wrappers for applications such that an exception can be caught by the wrapper and returned as an error when it is known a priori that the error is handled gracefully, while an exception is not.

In summary, the work performed under this contract has significantly advanced the state-of-the-art in a key area or security and dependability of Win32 systems: software robustness. We've developed and delivered three distinct technologies for analyzing and improving the robustness of Win32 systems under unusual conditions associated either with malicious attack or misbehaving operating system functions.

4. Statement of Work References

This section references each of the items in the original statement of work, and identifies sections of this report that discuss the completion of each item.

4.1 The contractor shall accomplish the following:

4.1.1 Input Generation Language

RIDDLE, NetHose, and the Grammar Generator each make use of a language for specifying the type of input that is produced for testing. The language that RIDDLE requires is used in the configuration files to specify the data generators that will be used, and is discussed in sections 2.1.3 and 2.1.4.2. The NetHose specification language is touched on briefly in this report, and is explored more thoroughly in the System User's Documentation. The language used by the grammar generator is based on Backus-Naur Form, and is described in section 2.1.4.5.2 of this report.

4.1.2 Input Generation and Perturbation Module

The tools mentioned in the previous paragraph (4.1.1) each implement an input generation and perturbation module that takes input in the form of the configuration language, and produces output that can be used to test applications. This modules relies on the use of data generators, which are described in sections 2.1.3 and 2.2.4.

4.1.3 Security Assertion Language

The security assertion language used by the testing utilities that we've developed has been built into the tools themselves. We did not have the need to define a very complex SAL for the type of testing that we performed. The data that we gathered was fairly coarse-grained, and we were able to build the necessary checks into the security watchdog module (described in the following paragraph), without the need for a separate security assertion languge.

4.1.4 Security Watchdog Module

RIDDLE implements a security watchdog that is capable of detecting how an application or function terminates (section 2.1.4.2). This watchdog is capable of differentiating between the normal termination of an application or function, and termination that has resulted due to an error or robustness violation.

The security watchdog used by NetHose is used to determine the status of the remote machine being tested. The watchdog process periodically checks to make sure that the network stack of the machine under test is still operating normally (see section 2.2.5).

4.1.5 Execution Manager

The execution manager used by RIDDLE and NetHose (see sections 2.1.4.2 and 2.2.5) combines the input generation and perturbation module with the security watchdog module to provide fully automated collection and analysis of data. The execution manager serves as the core component of these two utilities.

4.1.6 COTS Experimentation

Section 2.1.4.3.3 describes the results of testing the Win32 API with RIDDLE. Section 2.1.4.4.3 describes the results of testing native Microsoft command-line utilities and ported Cygnus GNU utilities with RIDDLE. The grammar generator was used to test both command line utilities (section 2.1.4.5.3) and a variety of web browsers and FTP servers (section 2.1.4.5.4).

NetHose was used to test the network stack of both the Windows NT and Windows 95 operating systems (section 2.2.7).

4.1.7 Deliver and demonstrate the CVA prototype system at RL / IWT

The time and date for this meeting has not yet been determined.

4.1.8 Reports and Documentation

The reports and documentation produced for this product satisfy the requirements as specified in the contract.

4.1.9 Software

All software developed under this contract will be delivered to the government according to SOW item 4.1.9 along with this report.

5. Papers and Presentations

A.K. Ghosh, M. Schmid, "Techniques for Evaluating the Robustness of Windows NT Software", to appear in the DARPA Information Survivability Conference and Exposition (DISCEX) 2000, January 25-27, 1999, Hilton Head, SC.

A.K. Ghosh, M. Schmid, "An Approach to Testing COTS Software for Robustness to Operating System Exceptions and Errors", 10th International Symposium on Software Reliability Engineering (ISSRE-10), November 1-4, 1999, Boca Raton, FL.

A.K. Ghosh, M. Schmid, F. Hill, "Wrapping Windows NT Software for Robustness", in Proceedings of the 29th International Fault Tolerant Computing Symposium (FTCS-29), June 15-16, 1999, Madison, WI, pp. 344-347.

A.K. Ghosh and J.M. Voas, "Inoculating Software for Survivability", in Communications of the ACM, Volume 42, No. 7, July 1999, pp. 38-44.

A.K. Ghosh, F. Hill, and M. Schmid, "NetHose: A Tool for Finding Vulnerabilities in Network Stacks", short talk presented at 1999 IEEE Symposium on Security and Privacy, May, 1999, Oakland, CA.

A.K. Ghosh, M. Schmid, and V. Shah, "Testing the Robustness of Windows NT Software", in Proceedings of the 9th International Symposium on Software Reliability Engineering (ISSRE-9), November 4-7, 1998, Paderborn, GE, pp. 231-235.

A.K. Ghosh and M. Schmid, "Wrapping Windows NT Binary Executables for Failure Simulation", in Proceedings of Fast Abstracts and Industrial Practices in the 9th International Symposium on Software Reliability Engineering (ISSRE-9), November 4-7, 1998, Paderborn, GE, pp. 7-8.

A.K. Ghosh, V. Shah, and M. Schmid, "An Approach for Analyzing the Robustness of Windows NT Software", in Proceedings of the 21st National Information Systems Security Conference (NISSC-21), October 5-8, 1998, Crystal City, VA, pp. 383-391.

M. Schmid and F. Hill, "Data Generation Techniques for Automated Software Robustness Testing", in Proceedings of the 16th International Conference and Exhibition on Testing Computer Software (ICTCS'99), June 16-18, 1999, Bethesda, MD.

M. Schmid, "The Failure Simulation Tool", A Poster Session, Presented at the 1999 USENIX Windows NT Symposium, July 12-14, 1999, Seattle, WA.

[1] Gen. John J. Sheehan. A commander-in-chief's view of rear-area, home-front vulnerabilities and support options. In *Proceedings of the Fifth InfoWarCon*, September 1996. Presentation, September 5.

[2] B.P. Miller, L. Fredrikson, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32-44, December 1990.

[3] B.P. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin, Computer Sciences Dept, November 1995.

[4] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz. Comparing operating systems using robustness benchmarks. In *Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems*, pages 72-79, October 1997.

[5] A. Ghosh, M. Schmid, V. Shah. An Approach for Analyzing the Robustness of Windows NT Software. In *Proceedings of the 21st National Information Systems Security Conference*, October 5-8, 1998, p. 374-382. Crystal City, VA.

[6] A. Ghosh, M. Schmid, V. Shah. Testing the Robustness of Windows NT Software. To appear in the *International Symposium on Software Reliability Engineering (ISSRE'98)*, November 4-7, 1998, Paderborn, GE.

[7] J. Richter. *Advanced Windows, Third Edition.* Microsoft Press, Redmond Washington, p. 529, 1997.

[8] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-over flow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63-78, San Antonio, TX, January 1998.

[9] E.H. Spafford. The Internet Worm Program: An Analysis. *Computer Communications Review*, 19(1):17-57, January 1989.

[10] W. Stevens. *TCP/IP Illustrated, Volume 1.* Addison Wesley Longman, Inc, Reading, MA. pp. 34-37, 144-150, 1994.

[11] M. Binderberger. Re: Navy Turns to Off-The-Shelf PCs to Power Ships (RISKS-19.75). *RISKS Digest*, 19(76), May 25, 1998.

[12] G. Slabodkin. Software Glitches Leave navy Smart Ship Dead in the Water, July 13 1998. Available online: www.gcn.com/gcn/1998/July13/cov2.htm.

[13] J.M. Voas. COTS: The Economical Choice? *IEEE Software*, 15(2), March / April 1998.

[14] N.P. Kropp, P.J. Koopman, and D.P. Siewiorek. Automated Robustness Testing of Off-The-Shelf Software Components. In *Proceedings of the Fault Tolerant Computing Symposium*, June 23-25 1998.

[15] B. Beizer. *Software Testing Techniques*. Electrical Engineering / Computer Science and Engineering. Van Nostrand Reinhold, 1983.

[16] B. Beizer. *Black Box Testing*. Wiley, New York, 1995.

[17] G. Myers. *The Art of Software Testing*. Wiley, 1979.

[18] B. Marick. *The Craft of Software Testing*. Prentice-Hall, 1995.

[19] J. Duran and S. Ntafos. *An Evaluation of Random Testing*. IEEE Transactions on Software Engineering, SE-10:438-444, July 1984.

[20] J.M. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley and Sons, New York, 1998.

[21] C. Howell. Error Handling: When bad things happen to good infrastructures. In *Proceedings of the 2nd Annual Information Survivability Workshop*, pages 89-92, November 1998. Orlando, FL.

☆U.S. GOVERNMENT PRINTING OFFICE: 2002/710-038/10263

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.